RAND

The RAND Advanced Simulation Language Project's Declarative Modeling Formalism (DMOD)

Jeff Rothenberg, Sanjai Narain

National Defense Research Institute

The research described in this report was sponsored by the Advanced Research Projects Agency under RAND's National Defense Research Institute, a federally funded research and development center supported by the Office of the Secretary of Defense, the Joint Staff, and the defense agencies, Contract MDA903-90-C-0004.

Library of Congress Cataloging in Publication Data

Rothenberg, Jeff
The RAND Advanced Simulation Language project's declarative modeling formalism (DMOD) / Jeff Rothenberg, Sanjai Narain.
p. cm.
"Prepared for the DoD's Advanced Research Projects Agency."
"MR-376-ARPA."
Includes bibliographical references.
ISBN 0-8330-1555-9
1. United States—Armed Forces—Transportation.
2. Transportation, Military—Computer simulation. 3. Digital computer simulation. 4. RAND Advanced Simulation Language (Project) I. Narain, Sanjai, 1957– . II. United States. Advanced Research Projects Agency. III. Title.
UC273.R68 1994
355.4—dc20 94-20774
CIP

RAND is a nonprofit institution that seeks to improve public policy through research and analysis. RAND's publications do not necessarily reflect the opinions or policies of its research sponsors.

Published 1994 by RAND
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90407-2138
To order RAND documents or to obtain additional information, contact Distribution Services: Telephone: (310) 451-7002; Fax: (310) 451-6915; Internet: order@rand.org.

RAND/MR-376-ARPA

RAND

The RAND Advanced Simulation Language Project's Declarative Modeling Formalism (DMOD)

Jeff Rothenberg, Sanjai Narain

Prepared for the
Advanced Research Projects Agency

National Defense Research Institute

Approved for public release; distribution unlimited

Preface

This report discusses research performed by the RAND Advanced Simulation Language ("RASL") project. This research was conducted for the Transportation Planning and Scheduling Initiative of the DoD's Advanced Research Projects Agency (ARPA) within the Applied Science and Technology Program of RAND's National Defense Research Institute (NDRI), a federally funded research and development center sponsored by the Office of the Secretary of Defense, the Joint Staff, and the defense agencies.

The objective of the RASL project has been to develop techniques to answer questions that go beyond the *"What if . . . ?"* capabilities of traditional simulation: questions such as "Can a given event ever happen?" "Under what conditions will an event happen?" or "How can a desired result be achieved?" In particular, the project is attempting to integrate knowledge-based simulation and planning to answer such questions in the strategic mobility domain. This report should be of interest to transportation planners, modelers, and researchers investigating new approaches to simulation and modeling. The discussion of strategic mobility is intended to introduce modelers to the broad outlines of this domain, while the discussion of modeling techniques (particularly the "logic programming" foundations of the modeling approach being pursued by the RASL project) is intended to introduce domain specialists to this technology. The technical details of the discussion will be of interest primarily to modeling methodologists with a background in logic programming.

Contents

Figures

Summary

This report discusses research by the RAND Advanced Simulation Language ("RASL") project. The objective of this research project has been to develop knowledge-based techniques that integrate simulation and planning to answer strategic mobility questions that go beyond the *"What if . . . ?"* capabilities of traditional simulation: questions such as "Can a given event ever happen?" "Under what conditions will an event happen?" or "How can a desired result be achieved?"

The strategic mobility planning problem encompasses a wide range of issues concerning which transportation assets (ships, planes, etc.) to acquire and how to use them to transport personnel and materiel to satisfy mission objectives. The RASL project is researching new modeling techniques that will allow simulation and planning to be performed in an integrated fashion, using a single, underlying model of the strategic mobility domain. This project has developed a new declarative modeling formalism (DMOD) for modeling and reasoning about dynamic systems. DMOD has four main features: First, it is based on the causality relation between events; the intuitive nature of this relation provides good heuristics both for structuring models and for proving their properties. Second, it can be used to model *continuous* as well as discrete time, eliminating logical errors that arise from the inappropriate choice of a discrete time step; similarly, DMOD can model continuous or discrete changes in state. Third, DMOD can be used to model situations in which the effects of an event depend not only on its past but also on its future; this not only simplifies the modeling of discrete systems but also allows the modeling of hybrid systems, i.e., those whose state contains both discrete and continuous parameters. Finally, the view of causality introduced by DMOD allows a wide range of intuitions about causality to be formalized using definite clauses (i.e., logic programs). This allows the expressive power and simple proof theory of definite clauses to be exploited to simplify both model development and proofs of the properties of models. DMOD can be thought of as an attempt to formulate a logical description of the event-scheduling view of discrete-event modeling.

1. Introduction

This report discusses research by the RAND Advanced Simulation Language (RASL) project. The objective of this research project has been to develop knowledge-based techniques that integrate simulation and planning to answer strategic mobility questions that go beyond the *"What if . . . ?"* capabilities of traditional simulation: questions such as "Can a given event ever happen?" "Under what conditions will an event happen?" or "How can a desired result be achieved?"

After providing background on strategic mobility planning and the RASL project, we present a revised descriptic n of the declarative modeling formalism (DMOD) that has been developed by this project. DMOD is a new, declarative framework for modeling and reasoning about dynamic systems, which can be thought of as an attempt to formulate a logical description of the event-scheduling view of discrete-event modeling (as, for example, discussed in Fishman [1973] and in Zeigler [1984]).

DMOD has four main features. First, it is based on the causality relation between events. The intuitive nature of this relation provides good heuristics both for structuring models and for proving their properties. Second, DMOD can model *continuous* as well as discrete time, eliminating the possibility of logical errors arising from an inappropriate choice of discrete time step as well as the need to manage the ticking of a simulation clock. In addition, it can be used to model continuous changes in state, as well as discrete changes. Third, DMOD can be used to model situations in which the effects of an event depend not only on its past but also on its future. This not only simplifies modeling of discrete systems but also allows the modeling of hybrid systems, i.e., systems whose state contains both discrete and continuous parameters, as will be made clear below. Most temporal calculi are intended only for modeling discrete systems, e.g., Petri Nets [Peterson, 1977], Temporal Logic [Pnueli, 1977; Manna & Pnueli, 1981], finitely recursive processes [Inan & Varaiya, 1987], finite automata [Kurshan, 1992], Real-Time Logic [Jahanian & Mok, 1986], or L.0 [Cameron et al., 1990; Ness, 1990]. Finally, DMOD allows a wide range of intuitions about causality to be formalized using definite clauses (i.e., logic programs). This allows the expressive power and simple proof theory of definite clauses to be exploited to simplify both model development and proofs of the properties of models.

The next section provides background on the RASL project and its problem domain, strategic mobility. Section 3 introduces DMOD and its view of causality. Section 4 presents examples of DMOD programs. Section 5 presents two algorithms for computing event occurrences. Section 6 outlines an approach for formally proving temporal properties of domains modeled using DMOD. Section 7 discusses the relationship of DMOD to previous work. Section 8 outlines the status of this research and future directions.

2. The RASL Project and Strategic Mobility Planning

Computer simulation is used throughout the Department of Defense (DoD) for analysis, training, and decision support; yet traditional simulation can answer only *"What if . . . ?"* questions (i.e., "What would happen if . . . ?"). In particular, most existing transportation models for strategic mobility analysis can answer only questions such as: "What would happen if we attempted to transport a given set of movement requirements using a given set of lift assets?" Yet many important questions go beyond *"What if . . . ?"* such as "Can unit X ever arrive before unit Y?" "Under what conditions will unit Y be late?" or "How can a desired closure profile be achieved?" (i.e., goal-oriented or planning questions). Such questions compose the bulk of the analyses that are performed by strategic mobility planners in organizations such as the Logistics Directorate of the Joint Staff. Answering such questions requires reasoning about models themselves, which is all but impossible for traditional simulations.

The limitations of most existing strategic mobility models [Schank et al., 1991] derive from limitations in their underlying modeling technology. With few exceptions, the transportation community has adopted simulation as the modeling method of choice for analyzing strategic mobility questions. This choice is largely due to the fact that traditional goal-oriented optimization techniques (such as mathematical programming) are generally unable to handle the huge data sets that are typically involved in mobility analyses since they must consider all of the data at once. Simulations, on the other hand, since they work iteratively, can handle arbitrarily large data sets.

Since the only transportation questions that are directly addressed by traditional mobility simulation models are essentially feasibility questions (i.e., "Can the available lift deliver the given movement requirements on time?"), these models do not serve the needs of many mobility planners. Many of these models do not even directly address scheduling questions, such as "How should a set of available vehicles be used to move a given set of movement requirements?" Yet studies such as the CMMS (Congressionally Mandated Mobility Study) and its replacement, the MRS (Mobility Requirements Study), are concerned with questions such as what future mixes of lift to acquire and how best to preposition materiel to address future threats. In an attempt to answer such questions, existing mobility models are often run hundreds of times, in effect searching for answers; yet such blind search techniques are highly ineffective and rarely yield satisfactory answers. Moreover, much of long-range planning (as discussed in detail below) involves building and analyzing plans, rather than simply testing their feasibility. Finally, mobility analysis must include an assessment of how close real-world methods and procedures can come to achieving theoretically optimum performance, in order to develop plans that are realistically executable.

In summary, strategic mobility planning involves much more than simply answering feasibility ("*What if... ?*") questions. In addition, it goes beyond simply building schedules or even plans: It is concerned with analyzing alternatives, evaluating trade-offs, understanding the realistic constraints on optimality, and constructing plans that are robust under uncertainty across a range of scenarios. None of the existing modeling and analysis techniques come close to addressing these needs. It has been the intent of the RASL project to explore and develop new techniques to help overcome this shortfall in modeling technology.

Overview of Strategic Mobility

Strategic mobility provides the United States with leverage for meeting a wide range of threats around the world. The changing world situation is rapidly transforming these threats, resulting in increasing emphasis on the need for fast, flexible response to a wide range of unpredictable, low- to medium-intensity conflicts. There are also indications that the military may increasingly be called upon to provide humanitarian relief support in addition to meeting traditional threats. At the same time, budget reductions and evolving political constraints are expected to make it increasingly difficult for the United States to preposition significant forces and equipment in close proximity to many areas of expected need. These conditions imply that strategic mobility—defined broadly as the ability to move forces and equipment in a timely and cost-effective manner to wherever they are needed—will become an increasingly important factor.

Although better ships and planes (generically called "lift") and supporting facilities might theoretically increase strategic mobility capacity, developing and maintaining such capital-intensive systems is very expensive. Improving the planning and management of strategic mobility, on the other hand, can greatly improve the ability to respond effectively to unpredictable needs, with relatively low capital investment. Furthermore, the resulting technology should transfer readily to other domains, both military and commercial. The planning and management of strategic mobility is therefore a promising area for research, offering a high degree of leverage for meeting evolving requirements in a changing world.

The Dimensions of Strategic Mobility Planning

Strategic mobility is an important aspect of virtually every military plan. The formation of a concept of operations or the elaboration of a specific course of action must always be conditioned and constrained by the availability of suitable transportation assets. Although the operational aspects of a plan are often thought of as quite separate from its transportation aspects, the two are intimately related. The transportation requirements of a plan are derived from its operational requirements, while the operational aspects of the plan are constrained by the feasibility of its transportation requirements. In this sense, strategic mobility planning is always "situated" in the context of operational planning.

At one extreme, the U.S. Transportation Command (USTRANSCOM) or one of the Transportation Component Commands (TCCs), such as the Air Mobility Command (AMC), may be concerned with the specific loading and scheduling of cargo on aircraft or ships to fulfill the mobility requirements of a specific operations plan (OPLAN). At the other end of the spectrum, the Joint Staff (JS) or the Office of the Secretary of Defense (OSD) may be concerned with specifying and procuring a collection of transportation assets that satisfy a range of scenarios, corresponding to a range of expected needs; in such cases, strategic mobility planning is situated in the context of this range of scenarios rather than in that of a single, specific plan. In between these extremes lies the complex process of developing transportationally feasible OPLANs. This entire range of activities is included in the term "strategic mobility planning" as it is commonly used.

It is useful to distinguish several specific types of strategic mobility planning, each of which normally occurs at a different level [Schank et al., 1991]. Resource planning, as performed at the JS or OSD level, is concerned with the long-range planning and procurement of transportation assets. Deliberate planning, performed well in advance by the Commanders in Chief (CINCs) with input from USTRANSCOM, the TCCs, and the JS, produces OPLANs to meet specific expected threats. Crisis action planning is analogous to deliberate planning but is performed under the time constraint of an evolving crisis situation—which may or may not already have been anticipated by deliberate planning—and leads into execution planning and execution. Execution planning produces a detailed, executable plan for *how* to move the cargoes required by a specified OPLAN.

Note that execution planning includes both the initial planning performed before execution is begun (sometimes *immediately* before) and dynamic replanning for each period as execution proceeds. The latter implies that execution planning is not really distinct from execution: The need to replan in a "situated" and "reactive" mode requires real-time access to operational information as execution proceeds.

Strategic Mobility Planning Questions

At each of the levels at which strategic mobility planning is performed, various questions may be asked. These can be divided roughly into two categories: *requirements* questions and *assessment* questions. Requirements questions are of the form "What transportation assets are needed to do a given job?" i.e., "What transportation assets are required to deliver a particular set of cargoes by a particular set of delivery dates, given a particular set of scenario assumptions?" In addressing these questions, movement requirements—the set of cargoes to be moved, along with their required delivery dates—are taken as given. Since there is often a choice of various combinations of lift (e.g., different numbers and kinds of ships or planes), requirements questions may include a cost function that constrains the optimum mix of assets. Whether or not such a cost function can be defined, requirements questions involve continual trade-offs among assets and capabilities. These questions are asked at high levels (by JS or OSD), in the course of procuring transportation assets and configuring the overall strategic mobility system, and at lower levels (such as USTRANSCOM or AMC), when specifying the lift assets needed to carry out a particular operation.

Assessment questions, on the other hand, are of the form "Given a set of movement requirements, a set of available transportation assets, and a set of scenario assumptions, what is the earliest that the various cargoes can be delivered?" i.e., "What can be done with the transportation that is available?" The resulting set of delivery dates is referred to as a "closure profile," so the assessment question can be rephrased as "What closure profile will result from using the available transportation assets to transport the given movement requirements?" One variant of an assessment question is a *feasibility* question, stated in terms of a required closure profile: In this case the question is "Given a set of movement requirements, a set of available transportation assets, and a set of scenario assumptions, is it feasible to achieve a given, *desired* closure profile?" The answer to a feasibility question is strictly "yes" or "no" although it is generally also important to provide insight into which constraints are critical.

In the process of answering assessment questions, it may also be necessary to answer *scheduling* questions of the form "Given a set of movement requirements, a set of available transportation assets, a set of scenario assumptions, and a required closure profile, *how* should the transportation assets best be used to deliver the cargo?" Scheduling can be thought of as a specific kind of planning, in which the activities to be performed (i.e., assigning cargoes to vehicles) are known in advance from the movement requirements. In the simplest case, scheduling simply assigns an ordering to these activities along with starting times for each one. Assessment questions are often answered by constructing a schedule (thereby answering a scheduling question at the same time). However, answering an assessment question does not *necessarily* involve answering a scheduling question: The feasibility of a plan can often be determined without constructing a schedule at all, for example, by using aggregate measures (such as tons or passengers per day, week, or month) to compare required lift to available lift. Nevertheless, most of the questions addressed by USTRANSCOM and the TCCs require the construction of detailed schedules.

Impact of Recent World Events

There is widespread recognition that recent changes in the world are making strategic mobility increasingly important, as standing forces are reduced and threats and other demands become more diverse. Strategic lift itself will come under increased scrutiny as budgets are reduced, making strategic mobility planning even more important. USTRANSCOM, JS, OSD, and others will be increasingly pressured to make effective and efficient acquisition, allocation, and planning decisions. The RASL project is directly concerned with producing the kind of planning technology that is required for longer-range planning efforts such as requirements planning; we believe this emphasis has been unique within the ARPA planning and scheduling community, which has focused primarily on scheduling and short-range planning.

Much of traditional artificial intelligence planning research has been oriented toward developing plans for use by robots, i.e., plans that are intended to be put to immediate use (this includes "deliberative" planning systems whose results are intended for immediate execution). In contrast, long-range mobility planning is required to produce plans for people, i.e., plans that are to be used for analysis, modification, and insight as much as for eventual execution. This requires a shifting of focus away from detailed, highly optimized plans toward higher-level, realistic, and robust plans whose underlying assumptions and dependencies are explicit and easily comprehended. The future appears likely to be characterized by multiple, diverse, small-scale threats and other demands;

whereas these will require fast, responsive planning prior to execution, their diversity and uncertainty will require longer-range planning that is increasingly broad and robust.

The RASL project's goal has been to develop modeling methods that can answer precisely the kinds of questions that are increasingly being asked by mobility planners.

The Strategic Mobility Planning Problem

The strategic mobility planning problem can be stated in terms of a lack of technology support. Better tools are needed to help strategic mobility planners at all levels answer *requirements, assessment, feasibility,* and *scheduling* questions to improve resource planning, deliberate planning, and crisis action/execution planning.

The actual generation of plans or schedules is only one aspect of strategic mobility planning. It is equally important to be able to analyze plans and their consequences, make trade-offs among alternatives, evaluate the transportational feasibility of proposed courses of action, and configure mixes of transportation assets that are most likely to satisfy an expected range of scenarios. Furthermore, the incremental improvement and modification of plans to adapt to evolving threats and operational circumstances is of paramount importance; it is probably fair to say that, throughout much of the strategic mobility domain, *planning is replanning*.

Tools for strategic mobility planners should help them perform all of these activities in a coordinated way. This implies the need for a true synthesis of planning and simulation. This synthesis must first help generate plans and schedules and answer straightforward *"What if . . . ?"* questions, but it should also help answer a wide range of other questions that go beyond *"What if . . . ?"* (such as why a proposed plan is infeasible, which constraints are the most limiting ones in a given situation, what trade-offs might be made to improve a plan, or what transportation assets are needed to do a given job).

Answering planning questions that go beyond *"What if . . . ?"* requires *reasoning* about a plan and the reality within which it is situated. Simulation alone cannot answer such questions because the number of cases to be examined may be large or infinite. For example, a requirements question such as "What transportation assets are needed to do a given job?" cannot be answered effectively by simulation. Planning, constraint satisfaction, optimization, and simulation all play a part in answering such questions, but they must be part of an integrated reasoning capability based on a model of the strategic mobility planning environment.

To further motivate the need for such modeling capabilities, the next subsection discusses the technology that has heretofore been used in strategic mobility planning.

Existing Technology for Strategic Mobility Planning

Many strategic mobility models have been built—and are being built—to help the various organizations perform the strategic mobility planning tasks described above. However, neither resource planning nor requirements questions in general are fully supported by existing or planned strategic mobility models. Furthermore, only certain aspects of the deliberate planning process are supported by existing models, and crisis action planning is neither fully supported in its own right nor by carryover from deliberate planning [Schank et al., 1991]. (It is also widely acknowledged that appropriate and reliable data are lacking in many areas of strategic mobility, particularly for crisis action and execution planning.)

Strategic mobility models have generally been designed according to one of two approaches: optimization techniques or simulation. Over the years, various mathematical programming models have been built in an attempt to provide optimal answers to requirements and assessment questions. These efforts have suffered from a variety of limitations, including: (1) the difficulty of representing complex closure profile requirements (e.g., involving time windows) and mixtures of linear and integer constraints in a mathematical programming formalism, (2) the lack of transparency and explanatory capability of such models, and (3) the computational intractability of mathematical programs representing realistically large strategic mobility problems. Nevertheless, new algorithms (i.e., Karmarkar) and hardware (i.e., KORBX [Carolan, 1990], supercomputers, multiprocessors, etc.) have revived interest in this approach. Related techniques for constraint satisfaction [Fox et al., 1989; Sadeh & Fox, 1989] also appear promising.

The bulk of recent strategic mobility models, however, have been simulations. These models generally attempt to produce a constructive answer to an assessment question by producing a "good" schedule (thereby also answering the corresponding scheduling question). Given a set of cargoes, a set of transportation assets, and a desired delivery profile, these models attempt to construct a schedule that meets the desired profile by (roughly speaking) simulating the heuristics that human schedulers use. For example, such models may try to assign the cargo with the earliest required delivery date to the fastest readily available vehicle and to assign the cargo with the next required delivery date to the next-available, next-fastest vehicle, etc. In general, these models start with an initial set of conditions and assumptions and answer the question "*What* would happen *if* the transportation system were faced with this situation?" Like most simulations, these models do not attempt to go beyond such "*What if . . . ?*" questions: They simply trace one path of the behavior of the (simulated) world for each set of initial conditions.

The main attraction of these simulation models is that they proceed iteratively, scheduling one cargo at a time, which allows them to handle very large sets of movement requirements. Since they simulate the behavior of the transportation system in some detail, they achieve credibility (or "face validity") to the extent that their behavior appears reasonably close to the way strategic mobility planners expect the real world to behave. However, these simulations work in only one direction: That is, they can directly answer only "*What if . . . ?*" questions. Using such models to answer requirements questions, for example (such as "What mix of lift assets would best satisfy a given closure profile?"), often requires many iterations and still cannot guarantee finding an answer.

In addition, the complexity and nondeclarative nature of these programs makes them hard to understand and validate. In particular, it is often difficult to ascertain whether their heuristics truly model the way real-world schedulers behave or whether they have been chosen simply for reasons of programming convenience.

To summarize, neither traditional optimization nor simulation fully answer the needs of strategic mobility planning. Optimization works backward to produce a plan (or at least a solution) from a goal, but it provides little insight into the feasibility of a preexisting, proposed plan, and it has little explanatory or predictive power. Furthermore, when optimization fails to find a solution, it merely concludes that the problem is infeasible, without explaining why. Simulation works forward from an initial state but can only grope toward a desired goal, let alone optimality. The next section elaborates the notion of an integrated simulation and planning environment that would answer a much larger subset of the questions posed by strategic mobility.

The ARPA/Rome Laboratory Transportation Planning and Scheduling Initiative focuses on a subset of the strategic mobility problem to stimulate research that can improve scheduling in execution planning. It emphasizes the production of schedules and transportation plans but implicitly also calls for new technology to support the entire planning process discussed above. This requires the close integration of simulation and planning.

To significantly affect strategic mobility planning, any new technology should aid planners in answering both requirements questions and assessment questions (including feasibility and scheduling questions), and it should provide insights into trade-offs, constraints, and the overall behavior of the strategic mobility system. This requires a synthesis of planning and simulation that can help generate plans and schedules, answer *"What if . . . ?"* questions, and answer a wide range of other questions that go beyond *"What if . . . ?"* Many of these questions can be answered only by reasoning about a plan and the reality within which it is situated. The next section discusses the details of the RASL project's DMOD formalism, which is an attempt to provide such capabilities.

3. Declarative Modeling to Integrate Simulation and Planning

Recent work in simulation at RAND has evolved from a long history of research in the methodology of modeling. One of the first object-oriented simulation languages (ROSS) was developed at RAND in the early 1980s [McArthur et al., 1984; McArthur et al., 1985; Klahr, 1985]. A number of application models were subsequently built using ROSS, and it was extended in a number of ways, both at RAND and elsewhere [Klahr et al., 1982; Klahr et al., 1984; Callero et al., 1985; Nugent, 1983; Nugent & Wong, 1986; Hilton, 1987]. This experience led to the recognition of a number of major shortcomings in the object-oriented simulation paradigm [Rothenberg, 1986; Cammarata et al., 1988]. As a result of this work, the Knowledge-Based Simulation project (KBSim) was created to explore a number of extensions and alternatives to object-oriented simulation [Rothenberg et al., 1989]. This led to the development of DMOD.

DMOD is essentially a rule-based modeling formalism in which a model is represented by a causality relation among events. Causality goes beyond simply defining temporal relations and constraints: It models the intuitive notion that events have causes and effects within a given context. The representation of causality in DMOD has been developed to serve the needs of simulation, in contrast to related methods of temporal reasoning [Allen, 1984; Hanks & McDermott, 1987; Shoham, 1987]; the DMOD representation of causality appears to be an effective way to build a large class of models while providing a formal basis for discrete-event simulation. Using this representation, simulation in DMOD consists of computing a *history* of those events that actually occur, starting from a set of initial events and state values. This approach overcomes many fundamental shortcomings of traditional discrete-event simulation, notably the problem of event cancellation, which requires the undisciplined deletion of events that have been scheduled on an event queue when subsequent events preempt them.

DMOD is event oriented, which eliminates a number of artifacts of the object-oriented paradigm, such as the need to model all actions as being intentionally caused by specific, single objects. However, DMOD incorporates many of the key advantages of object-oriented modeling as well, including encapsulation and the ability to represent inheritance hierarchies.

Unlike most simulation methodologies, DMOD does not represent state variables by explicit storage cells, but instead expresses each state value as a function of time and the events in the history. In this way, state is considered to be piecewise continuous: The history of events serves as an efficient representation of discontinuous state changes, whereas state changes in between events are represented by continuous functions of time. This approach unifies the treatment of discrete and continuous state and time and allows state to be computed on demand since it can always be derived from the event history. This enables new state variables and ad hoc queries to be defined dynamically—even after running a simulation—without rewriting, reinstrumenting, or even rerunning the model. It also eliminates many errors that arise from the failure to maintain the consistency of stored state in traditional simulations. In practice, state is cached to improve performance while retaining these advantages.

DMOD separates causality from its enforcement by stating causal rules declaratively: All such rules are processed by a formal interpreter rather than specifying their own behavior. This separation eliminates the ad hoc causality that appears in most object-oriented and traditional simulations. In addition, it facilitates reasoning about the causality relationships in a model.

One of the major functional advantages of DMOD over other simulation methodologies is its potential to support reasoning in order to answer questions that go beyond "*What if. . . ?*" This has allowed us to implement novel analytic capabilities in a logistics model of the Wartime Theater Ammunition Distribution System (WTADS) [Schank et al., 1990] and a prototype strategic mobility model (SMM). For example, these models can trace the progress of a simulation in terms of causal chains, showing which events directly or ultimately caused (or were caused by) which other events. With some restrictions, causal chains can also be traced in the abstract, analyzing the model itself to see which events can ever (or must always) cause which other events. This can be used to prove "liveness" properties (such as the fact that a ship must eventually reach its destination, despite rerouting) or "safety" properties (such as the fact that no cargo will be delivered late), as discussed below. Similarly, one can ask why a simulation diverged from an expected course of behavior: With some guidance from the modeler, DMOD can determine what factors would have caused the expected behavior to occur. Finally, DMOD facilitates exploring various strategies for answering goal-directed questions, such as "How fast must a convoy move to guarantee that it cannot be intercepted?" or "How should a given lift asset be utilized to achieve a desired closure profile?"

The next subsections present the formal basis for reasoning about DMOD models and proving interesting and useful properties about them. The examples developed are very simple abstractions of problems from the transportation domain. These examples are intended to clarify DMOD's capabilities by avoiding unnecessary detail.

Declarative Modeling of Dynamic Systems

When modeling dynamic systems, it is useful (and not unreasonable) to make the following fundamental assumption [Misra, 1986], which can be thought of as defining the relationships among the intuitive concepts *behavior, history,* and *event*:

> **Fundamental assumption:** The *behavior* of a system up to time T can be computed from its *history* up to T, i.e., a complete record of all *event* occurrences in the system up to T.

This says that the behavior of a system is completely characterized by its history, i.e., by the events that occur in the system. (The definitions of *behavior, history,* and *event* are closely intertwined in this statement: A system's *behavior* consists of the *history* of *events* that occur in the system, whereas the *events* in the system are simply those occurrences that we consider to define its *behavior.*) Given this assumption, a *model* of a system, i.e., a formal description of the system sufficient for computing its behavior, need describe only how events occur in the system. Simulation of the system then consists of using this model to compute a history of the events that occur in the system under a given set of circumstances. Note that although the concept of "state" does not appear explicitly here, it is implicit in this fundamental assumption that the entire state of a system (that is, anything of interest about the system) is completely characterized by its history.

There are many ways of describing how events occur in a system, but one of the most intuitive—and useful—is to define a causality relation between events. Informally, the proposition causes(A,B) means that event A is responsible for, or brings about, event B. (For convenience we read causes(A,B) as "A causes B.") If the proposition occurs(X) means that event X actually occurs, and the proposition initial(X) means that event X is an initial event that has occurred before we begin modeling the system, then event occurrences can be computed by repeatedly applying the following rule:

$$\text{occurs(B) iff initial(B)} \vee [\exists \text{A} \mid \text{occurs(A)} \wedge \text{causes(A,B)}]$$

This states that event B occurs if and only if either B is an initial event or there exists an event A such that A occurs and A causes B. A definition of the relation causes(X,Y) for every pair of events X and Y in a system, therefore, constitutes a model of that system. For any domain in which we have good intuition about causality, it is straightforward to develop such a model by defining the causality relation among the events in that domain. In fact, the widely used event-scheduling view of the discrete-event modeling technique [Evans, 1988; Zeigler, 1984] can be regarded as being based on a similar, though less formal, notion of causality. DMOD can be viewed as an attempt to formalize the discrete-event modeling technique (see Section 7).

The above rule also yields a simple basis for answering questions about event occurrences. For example, to show that a (noninitial) event occurs, it is sufficient to show that at least one of its causes occurs. Similarly, to show that a (noninitial) event does *not* occur, it is sufficient to show that none of its causes occur. More general temporal properties can be proved if they can be expressed in terms of questions about event occurrences (see Section 6 for examples).

Central to the success of the above approach for computing event occurrences and proving temporal properties of systems is the simplicity of performing inference using the causality relation. Unfortunately, such inference can be quite difficult when modeling situations in which the effect of an event can depend not only on its past but also on its future: An important instance of this phenomenon is event *preemption*, in which an event is expected to occur at some future time but does *not* occur because some preemption condition is satisfied in the interim. For example, if at time T an alarm clock is set to ring after time Delay, then it is expected to ring at T+Delay; but if it is reset during the interval (T,T+Delay), then the ringing event is preempted.

Event preemption also arises in hybrid systems, i.e., those whose state contains both discrete and continuous parameters. Event occurrences initiate periods of continuous change; events are the boundaries between such periods. When such a period begins, the time at which it should end can be predicted, based on current information; however, if this information changes during the period, then this prediction can be invalidated. For example, suppose an aircraft A begins flight with constant velocity toward a fixed radar R. Then, based on the initial velocity and position of A, it can be predicted that R will detect A at some future point of time; however, if the velocity of A subsequently changes, then this prediction may be invalidated.

Modeling such phenomena using causality results in statements that contain conditions involving the time intervals between the causing and caused events. For example, for the alarm clock we have:

```
causes(set_alarm(T+Delay,T), alarm_sounds(T+Delay))
    if   ¬∃X∃Y | [T < X < T+Delay ∧ occurs(set_alarm(Y,X))].
```

Here set_alarm(Future,Now) denotes an event of setting the alarm at time Now to ring at time Future. (In this example, the alarm is set at time T to ring at time (T+Delay). Events can have arbitrarily many parameters specifying the characteristics of the event; we follow the convention that the time at which the event occurs, i.e., its time stamp—"Now" in this example—always appears as its last parameter.) Similarly, alarm_sounds(T) denotes an event of the alarm ringing at time T. The rule says that set_alarm will cause alarm_sounds if it is not preempted, i.e., if there is no time X in the interval (T,T+Delay) at which the alarm is reset to ring at some other time Y (even if Y happens to be equal to T+Delay, in which case this particular *instance* of the event alarm_sounds(T+Delay) will not occur, although a distinct, identical instance *may* occur). We say that the "preemption condition" for this rule is that set_alarm occurs again between T and T+Delay. The rule, therefore, says that setting the alarm at time T will cause it to ring at time T+Delay if the preemption condition is false.

Similarly, a causality rule for the aircraft/radar example would be:

```
causes(flies_toward(Aircraft,Radar,T), detects(Radar,Aircraft,T+TravelTime))
    if   travel_time(Aircraft,Radar,TravelTime,T)
         ∧ ¬∃X | [T < X < T+TravelTime
                     ∧ velocity(Aircraft,X) ≠ velocity(Aircraft,T)].
```

Here flies_toward(Aircraft,Radar,T) denotes the event of Aircraft beginning to fly toward Radar at time T, whereas detects(Radar,Aircraft,T+TravelTime) denotes the event of Radar detecting Aircraft at time T+TravelTime. The relation travel_time(Aircraft,Radar,TravelTime,T) is true if Aircraft, starting at time T and traveling at a fixed velocity, reaches Radar after TravelTime (this can be thought of as a function that computes TravelTime). The preemption condition is that at some time X in the interval (T,T+TravelTime) the velocity of Aircraft is different from its value at time T. (Logically, this condition is necessary but not sufficient, since the velocity may vary within this interval while still averaging out to the initial value; however, again, the particular *instance* of the detection event *will* be preempted if this preemption condition is true, even though a distinct, identical instance *may* occur.)

The difficulty of inferring causality from such statements is that preemption conditions are negative and occur in the antecedent of causality rules, so that the resulting rules are in full first-order logic. In general, it is difficult to control proof procedures for this logic. For example, the programming language Prolog, extended with "negation-as-failure," would go into an infinite loop if the alarm clock rule were submitted to it because of the mutual recursion between rules for occurs and causes: Ringing will occur if it is caused by setting, but setting causes ringing only if setting does not occur again in the interim; that is, *occurs* depends on *causes*, but *causes* depends on *occurs*.

To exercise greater control over the inference procedure we can interpret rules in a top-down manner. However, this leads to a difficult recursion. Suppose that we generate events in increasing order of time stamps and that for the alarm, the event set_alarm(T+Delay,T) has occurred. Then to determine whether alarm_sounds(T+Delay) occurs we have to evaluate the preemption condition, i.e., determine whether the alarm is reset in the time period (T,T+Delay). Since we have information only up to T, we can suspend the evaluation of this condition, generate information beyond T, and then resume. But this would call the inference procedure recursively. Exactly how is this suspension-recursion-resumption to be implemented—and proved correct—especially when preemption conditions can be arbitrarily complex? Note that if time is continuous, we cannot simply iterate over each point of time, since there are infinitely many such points.

One major contribution of DMOD is a way of reformulating causality rules so that conditions on time intervals have direct access to the sequence of events that occur in those intervals. This is made possible by a new view of causality. The resulting extra information allows considerable control over how efficiently preemption conditions are evaluated. This formulation allows a wide range of causality r ıles to be expressed using definite clauses, i.e., logic programs in a programming language such as Prolog. The expressive power of such programs can then be utilized to provide a wide range of other capabilities for modeling dynamic systems, such as rule-based inference or the solution of differential equations. Furthermore, the proof theory for definite clauses is considerably simpler than that for full first-order logic; this can be exploited to simplify proofs of various properties of models, as illustrated below.

DMOD: A New Technique for Modeling Dynamic Systems

We assume the usual first-order logic alphabet, consisting of an enumerably infinite list of variables, function and predicate symbols of all "arities" (number of arguments), and the logical connectives [Lloyd, 1984]. In addition, we make use of standard notational shorthands such as "w.r.t." for "with respect to." (While the development here is intended to be independent of any particular implementation, since DMOD has initially been implemented in the Prolog programming language, we retain consistency with Prolog syntax where feasible to avoid confusion. Furthermore, Prolog is used to provide concrete examples of how various required constructs can be implemented in a logic program.) The following definitions establish our terminology.

> **Dfn:** A *constant* is a symbol denoting a specific entity; we always write constants as beginning with a lowercase letter or as a number (e.g., x, positionOfX, 3.14159, 17).

> **Dfn:** A *variable* is a symbol that stands for an unspecified but single entity; we always write variables as beginning with an uppercase letter (e.g., X, Radar, Position).

> **Dfn:** A *term* is either a constant, a variable, or a structure of the form $f(X_1, \ldots, X_n)$, where f is a (constant) n-ary function symbol and $X_1, \ldots, X_n$ are (recursively) terms.

> **Dfn:** A *ground* term is a term in which no variables occur.

> **Dfn:** An *event* is a ground term of the form $f(a_1, \ldots, a_n, t)$ where f is a function symbol of arity n + 1 (referred to as an "event-defining" function symbol), the $a_1, \ldots, a_n$ are ground terms denoting arbitrary entities, and t is a ground term denoting a numeric time instant, interpreted as the time stamp of the event.

An event may denote an action. For example, the event sends(sender,message, receiver,t) denotes the action of sender sending message to receiver at time t. An event may also denote the act of a proposition becoming true. For example, the event touching(a,b,t) can denote the proposition that the distance between two objects a and b at time t is zero; when this proposition becomes true, a collision between a and b can be said to occur at t.

Note that although the definition of an event requires it to be ground, we can define the related notion of an "event-type" in which variables take the place of ground entities, thereby defining a class of events:

> **Dfn:** An *event-type* is a nonground term having the same form as an event, i.e., $f(X_1, \ldots, X_n, T)$, where f is an event-defining function symbol, but the $X_1, \ldots, X_n$ may be variables, and T may be a variable. This denotes a class of events involving the entities denoted by the $X_1, \ldots, X_n$ and occurring at the time denoted by T. For convenience, we may even substitute an expression for T (such as T+Delay), to constrain the allowed times of occurrence of events of this event-type.

For convenience, we will use the term "event" to mean either a specific event or an event-type, using the term event-type only when necessary to avoid confusion. In addition, we will conventionally denote the time stamp of any event E by TE.

Note that the event-defining function symbols must be chosen in such a way that the fundamental assumption (p. 14) is satisfied. Informally, this simply says that we must define all events that are of interest (to completely characterize the behavior of the system). For example, to compute the positions and velocities of a set of pool balls on an infinite pool table, the event function symbol *touching* might be sufficient; that is, this might be the only event-type of interest. Other possible conditions, such as *equilateral* [where equilateral(a,b,c,t) means that the positions of balls a, b, and c at time t are on the vertices of an equilateral triangle], might not be regarded as events of interest.

> **Dfn:** A logic program Ord *defines a linear* ordering between time stamps if it defines the binary relations time_less_than and time_equal between time stamps, satisfying the following:

Linear ordering restrictions:

(1) Effectiveness of time stamp ordering: In the presence of Ord, for any time stamps X and Y it is always possible to determine, in finite time, whether time_less_than(X,Y) succeeds. Similarly for time_equal(X,Y).

(2) In the presence of Ord, for any time stamps X and Y, one and only one of time_less_than(X,Y), time_less_than(Y,X), or time_equal(X,Y) succeeds.

(3) In the presence of Ord, for any time stamps X and Y, if time_less_than(X,Y) succeeds and time_less_than(Y,Z) succeeds then time_less_than(X,Z) succeeds.

(4) In the presence of Ord, time_equal(X,X) succeeds. Also, if time_equal(X,Y) succeeds then time_equal(Y,X) succeeds. Finally, if time_equal(X,Y) succeeds and time_equal(Y,Z) succeeds then time_equal(X,Z) succeeds.

Restriction (1) constrains Ord to be *effective* by requiring time_less_than and time_equal to be decidable. Restriction (2) is the standard trichotomy law; restrictions (2) through (4) ensure that any set of time stamps can be linearly ordered. For simplicity, we will normally omit the word "linear" and simply say that a logic program defines an "ordering," implying a linear ordering.

A concrete definition of time_less_than and time_equal can be developed in Prolog as follows: Let time stamps be only those Prolog terms denoting numbers. Then the standard "<" and "=" operations of Prolog satisfy (2) through (4). Since these operations as implemented in Prolog are also effective, they can be defined using logic programs and made to satisfy (1).

> **Dfn:** If Ord is a logic program defining an ordering between time stamps, and E and F are two events, then E is *strictly earlier* than F w.r.t. Ord if time_less_than(TE,TF) succeeds in the presence of Ord, where TE is the time stamp of E and TF is the time stamp of F. (When applying this and the following two definitions, Ord will not be denoted explicitly if it is clear from the context.)
>
> **Dfn:** If Ord is a logic program defining an ordering between time stamps, and E and F are two events, then E is *concurrent* with F w.r.t. Ord if time_equal(TE,TF) succeeds in the presence of Ord, where TE and TF are as above.
>
> **Dfn:** If Ord is a logic program defining an ordering between time stamps, then a finite or infinite sequence of events $S = E_0, E_1, \ldots$ is said to be *temporally ordered* w.r.t. Ord if for each i, $0 \leq i$, either E_i is strictly earlier than E_{i+1} or E_i is concurrent with E_{i+1} (whenever E_{i+1} exists).

As discussed on p. 14, defining a causality relation provides a convenient way of computing event occurrences; yet, if statements about causality are written in the obvious way, causality can be quite difficult to infer. We now show a way of viewing causality that allows such statements to be reformulated using definite clauses. To do this, we regard causality not as a binary relation but as a ternary relation between two events and a context. The context is a temporally ordered sequence of events (w.r.t. some fixed-ordering Ord). We call the new relation causal_connection. If it holds for two events, they are said to be causally connected in the context. Causal connectedness is similar to connectedness between nodes in a network. The same two events may be causally connected in one context but not in another. For example, consider the following events concerning a ball dropped from a height (the numeric arguments represent time stamps, in arbitrary units):

E_1=ball_dropped(0)

E_2=ball_caught(5)

E_3=ball_thrown_down(6)

E_4=ball_velocity_becomes_high(7)

E_5=ball_hits_ground(10)

Consider the two contexts $C_1 = [E_1,E_5]$ and $C_2 = [E_1,E_2,E_3,E_4,E_5]$. E_1 can be said to be causally connected to E_5 in context C_1, where dropping the ball causes it to hit the ground. But in context C_2, the potential causal connection between E_1 and E_5 is terminated by the appearance of E_2. Since the ball is caught (E_2), dropping it (E_1) is not what causes it to hit the ground (E_5). It is more natural to say that throwing the ball down (E_3) is what causes it to hit the ground in this case, i.e., that E_3 is causally connected to E_5 in context C_2.

Another way to understand causal connection is the following: Let C be a temporally ordered sequence of events. Pick out E and F in C such that E appears before F. If all events in C were to occur, then would E be said to be a cause of F? If so, then E is said to be causally connected to F in C. In general, the context C represents a history of events that actually occur.

For convenience, we actually define causal_connection as a four-ary relation (rather than ternary), involving two events E and F, and dividing the context into two pieces, HE and HEF, representing the history up to event E and the history between event E and event F, respectively (see Figure 1). (For convenience, when speaking of sequences of events, we will use the phrase "up to" to mean "up to but not including" whereas "through" will mean "up to and including," and we will use "between" to mean "not including either end point" whereas "from A through B" will mean "including both end points".)

We define causal_connection(E,HE,F,HEF) with arguments related as follows:

S is a *finite context*, i.e., a temporally ordered sequence of events through F. E appears before F in S, HE is the sequence of all events in S up to E, and HEF is the sequence of all events in S between E and F (using the above definitions of "up to" and "between").

If the condition causal_connection(E,HE,F,HEF) holds, E is said to be causally connected to F in context S.

RANDMR378-1

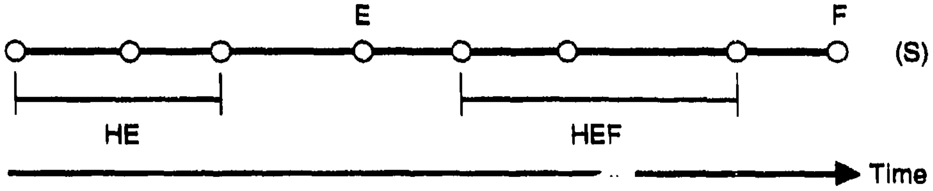

Figure 1—A History of Events

We next introduce several useful notational conventions:

> **Notation conventions:** As in Prolog, we use "[a,b,c]" to denote a list of items a, b, and c; "[x]" to denote a singleton list consisting of just the item x; "[]" to denote the empty list; and "[A | B]" to denote a list whose first item (or "head") is A and the remainder (or "tail") of which is the list B. Extending Prolog notation for convenience, we use "A•B" to denote the result of appending a list B to a list A. The operator "•" is associative. (Since an event E is not a list, we must write "HE•[E]" to denote the list consisting of HE followed by E.) Note also that the scope of a variable name is limited to the clause it appears in: Two occurrences of the same name (e.g., "E") in different clauses do *not* in general denote the same entity.

Let S represent the history of the system through F. Then, by the fundamental assumption above, HE•[E]•HEF is a complete record of all states and events in the past of F. HEF provides a way of resolving references to the future of E, up to F. For example, the aircraft/radar causality rule on p. 16 can be expressed using definite clauses as follows:

```
causal_connection(E,HE,F,HEF)
     if  E=flies_toward(Aircraft,Radar,T),
         F=detects(Radar,Aircraft,T+TravelTime),
         travel_time(Aircraft,Radar,TravelTime,HE•[E]),
         velocity(Aircraft,CurrentVelocity,HE•[E]),
         velocity_unchanged(Aircraft,CurrentVelocity,HE•[E],HEF).
```

where:

```
velocity_unchanged(Aircraft,CurrentVelocity,PastHist,[]).

velocity_unchanged(Aircraft,CurrentVelocity,PastHist,[X | RestOfFuture])
     if  velocity(Aircraft,VelocityAtX,PastHist•[X]),
         VelocityAtX = CurrentVelocity,
         velocity_unchanged(Aircraft,CurrentVelocity,PastHist•[X],RestOfFuture).
```

The first rule states that E is causally connected to F in the context HE•[E]•HEF•[F] if E is the event flies_toward(Aircraft,Radar,T), F is the event detects(Radar,Aircraft,T+TravelTime), TravelTime is determined by the relation travel_time, the velocity of Aircraft immediately after E has occurred is CurrentVelocity, and the velocity of Aircraft at any time during HEF (which is the future of E up to F) is unchanged from CurrentVelocity. Note that by the fundamental assumption, HE•[E] represents the state of the system at T (the time stamp of E), after E has occurred; therefore, the condition:

travel_time(Aircraft,Radar,TravelTime,T)

from the original causality rule (p. 16) can be replaced here by:

travel_time(Aircraft,Radar,TravelTime,HE•[E]).

Since we assume that velocity can change only at event boundaries, the first clause of the rule for velocity_unchanged says that if the list of future events is empty, then the Aircraft's velocity does not change. The second clause of this rule says that the Aircraft's velocity does not change over the list of future events, provided its velocity after the first such event (X) is the same as CurrentVelocity and the velocity does not change over the rest of the future events.

Of crucial importance here is that the condition from the original causality rule (p. 16):

$$\neg\exists X \mid [T < X < T+\text{TravelTime} \wedge \text{velocity (Aircraft,X)} \neq \text{velocity (Aircraft,T)}]$$

is replaced by:

velocity_unchanged(Aircraft,CurrentVelocity,HE•[E],HEF).

This is advantageous since the first of these forms represents a search over a potentially continuous interval of time, whereas the second is a computationally much simpler iteration over a finite sequence of events.

We are now in a position to define a DMOD program:

> **Dfn:** A logic program P is *a DMOD program* if it defines causal_connection, defines an ordering between time stamps, and satisfies the following restriction:
>
> > **DMOD effectiveness:** For any ground terms E,HE,HEF, it is always possible to determine, in finite time, the set of ground terms F such that causal_connection(E,HE,F,HEF) succeeds in the presence of P. Furthermore, this set is always finite.

This restiiction is needed to ensure the effectiveness of the algorithm for computing a history from a DMOD program. It can be satisfied by writing terminating logic programs (which, it must be admitted, is sometimes easier said than done).

4. Examples of DMOD Programs

This section shows examples of actual DMOD programs. These are presented in pseudo-Prolog for concreteness. In the following, we assume that a fixed ordering between time stamps has been supplied. The definition of causal_connection can refer to this as needed. We also assume that when ground terms X and Y denote numbers, the following two clauses are present in the ordering:

```
time_less_than(X,Y) if X < Y.

time_equal(X,X).
```

where "<" has the obvious meaning. The DMOD effectiveness restriction (above) is satisfied by ensuring that each DMOD program terminates. (DMOD programs rely on the logic of definite clauses and SLD-resolution,[1] which is described in Hill [1974] and in Lloyd [1984]. Formally, ensuring DMOD effectiveness requires ensuring that for any ground terms E,HE,HEF and for variable F, all SLD-derivations starting with the goal:

```
G=causal_connection(E,HE,F,HEF)
```

and using the "leftmost computation rule" are finite. This in turn guarantees that the set of all F such that the goal G succeeds can be obtained from the successful SLD-derivations in finite time.)

Reference to the Past

In causal_connection(E,HE,F,HEF), the term HE represents the past of E, the history of all events that have occurred up to E. By the fundamental assumption, this provides a complete characterization of the behavior of the system up to E. This allows causal_connection to refer to any required state in the past of E, by using HE. In contrast, traditional, state-oriented simulation approaches can refer to the past only by making the required history an explicit part of the current state; this makes their representation of current state large, unwieldy, and difficult to keep up to date. The following example shows how HE provides

[1] SLD is an acronym meaning selecting a literal, using a linear strategy restricted to definite clauses.

access to the past in a DMOD rule to make an answering machine answer after four rings.

```
causal_connection(E,HE,F,HEF)
    if    E = rings(T),
          F = answer(T),
          member(rings(T - 1),HE),
          member(rings(T - 2),HE),
          member(rings(T - 3),HE).
```

Where member has the obvious meaning: member(Item,List) is true if Item is a member of List. This can be defined recursively as:

```
member(A,[A | B]).

member(A,[U | V]) if member(A,V).
```

Here the delay between two rings is arbitrarily modeled as one time unit. The causal_connection rule states that a ringing event at time T causes an answering event at time T, if previous ringing events at times T – 1, T – 2, and T – 3 are all members of the history HE of E.

An Alarm Clock Example

In Section 3, we presented a model of an alarm clock. We now show a DMOD rule that expresses the fact that setting an alarm clock at time T to ring at time T+Delay causes it to ring at time T+Delay, provided no subsequent resetting event occurs in between. This example illustrates DMOD's ability to model event preemption. The rule is:

```
causal_connection(E,HE,F,HEF)
    if    E = set_alarm(T+Delay,T),
          F = alarm_sounds(T+Delay),
          non_member(set_alarm(?,?),HEF).
```

As discussed previously (Section 3, p. 22), HEF provides access to the future of E, up to F. Preemption becomes a simple matter of checking whether any subsequent set_alarm event occurs in HEF. [The notation "?" here stands for an arbitrary term: It is analogous to the unnamed variable "_" (underscore) in Prolog; the term set_alarm(?,?) can be thought of as a template that will match any set_alarm event. Informally, non_member(X,L) is true if there is no instance of the event-type X in the list L, where occurrences of "?" in X are allowed to match corresponding terms in L in the obvious way. Developing a logic program for non_member is straightforward.]

A Communication Protocol

We now model a simple communication protocol in which a Sender sends a sequence of packets to a Receiver over an unreliable communication channel. Upon receipt of a Packet, Receiver sends an acknowledgment to Sender. If Sender does not receive acknowledgment for a Packet within a fixed amount of time, timeout, it sends a new copy of Packet. In DMOD we can write:

```
causal_connection(E,HE,F,HEF)
    if    E = send(Sender,Packet,Receiver,T),
          F = receive(Receiver,Packet,Sender,T+Delay),
          expected_time_to_arrive(Delay),
          not_lost(Packet,HE●[E]●HEF).
```

Where expected_time_to_arrive can be defined as:

```
expected_time_to_arrive(Delay)
    if    lower_bound_for_arrival(L),
          upper_bound_for_arrival(U),
          random(L,U,Delay).
```

The first rule states that if Sender sends a Packet to Receiver at time T, then Packet is received by Receiver after some Delay, provided it is not lost in between. The relation expected_time_to_arrive computes a random time between the lower and upper bound for Packet to arrive. The relation not_lost determines whether Packet is lost, given the history HE●[E]●HEF up to F. Information about the reliability of the channel can be encoded in the definition of not_lost. To retain information about how many times a particular packet has been resent before it is received, we represent each packet as a structured term packet(Pkt,N), where Pkt is the actual contents of the packet (which we will call the packet "body") and N is a "repeat-count" of how many times this same packet body has been sent [in the rule above, the variable Packet would be bound to an instance of the structured term packet(Pkt,N)]. Whenever Receiver receives a packet, it acknowledges it by sending an ack reply:

```
causal_connection(E,HE,F,HEF)
    if    E = receive(Receiver,packet(Pkt,N),Sender,T),
          F = send(Receiver,ack(Pkt,N),Sender,T).
```

This states that if Receiver receives packet(Pkt,N) from Sender, it sends a reply ack(Pkt,N) to Sender immediately. However, since the channel is unreliable, the sender must resend a packet if it does not receive an acknowledgment within a specified "timeout":

```
causal_connection(E,HE,F,HEF)
    if    E = send(Sender,packet(Pkt,N),Receiver,T),
          F = send(Sender,packet(Pkt,N+1),Receiver,T+Delay),
          timeout_delay(Delay),
          non_member(receive(Sender,ack(Pkt,N),Receiver,?),HEF).
```

This rule states that if a packet of the form packet(Pkt,N) is sent by Sender to Receiver at time T and no acknowledgment of the form ack(Pkt,N) is received by Sender before time T+Delay (where Delay is the amount of timeout), then a repeat packet—packet(Pkt,N + 1)—is sent by Sender to Receiver at time T+Delay. The repeat-count (N + 1) in the new packet keeps track of how many times this packet has been resent. (Note: An embedded arithmetic expression such as "N + 1," representing the value of an argument in a relation, is not allowed in Prolog, but its effect is easily obtained, and the notation used here is more natural than that required by Prolog.)

In the case where an acknowledgment is received by Sender, the next packet is sent with an initial repeat-count of zero:

```
causal_connection(E,HE,F,HEF)
    if    E = receive(Sender,ack(Pkt,N),Receiver,T),
          F = send(Sender,packet(NextPkt,0),Receiver,T),
          waiting_for_ack(Sender,Pkt,HE•[E]),
          buffer(Sender,[NextPkt | RmndrOfBuffer],HE•[E]).
```

This states that when Sender receives an acknowledgment ack(Pkt,N) from Receiver and Sender was waiting for acknowledgment of this packet, it immediately sends to Receiver the 0th copy of the next packet in its buffer. We represent the sender's buffer as a list of packet bodies, the first (leftmost) of which is the next packet to be sent (since [NextPkt | RmndrOfBuffer] represents a list whose first element is NextPkt and whose remainder is the list RmndrOfBuffer). We define the relation buffer(Sender,Bufr,H) in such a way that the buffer of Sender is the list Bufr immediately after the last event in a history H:

```
buffer(Sender,NewBuffer,HE●[E])
    if   E = request_to_send(Pkt,Sender,T),
         buffer(Sender,OldBuffer,HE),
         NewBuffer = OldBuffer●[Pkt].
buffer(Sender,NewBuffer,HE●[E])
    if   E = send(Sender,packet(Pkt,0),Receiver,T),
         buffer(Sender,[Pkt | NewBuffer],HE).
buffer(Sender,NewBuffer,HE●[E])
    if   non_member2(E,[request_to_send(Sender,?,?),send(Sender,?,?,?)]),
         buffer(Sender,NewBuffer,HE).
```

These rules define how the buffer of a sender changes with event occurrences for a history HE●[E]. The first rule says that if event E consists of a request (from some unnamed agent) asking Sender to send the packet body Pkt, then the new buffer (NewBuffer) immediately after E consists of Pkt appended to the end of whatever the old buffer (OldBuffer) was prior to E (immediately after HE). The second rule says that if event E consists of the Sender sending the 0th copy of the packet body Pkt, then the new buffer (NewBuffer) immediately after E consists of the buffer immediately after HE with the packet body Pkt removed from its head (that is, the old buffer was [Pkt | NewBuffer]). The third rule says that if E is neither of the above two kinds of events (that is, E is not a member of a list consisting of templates for request_to_send and send events), then the new buffer (NewBuffer) immediately after E is the same as that after HE. (The relation non_member2 is analogous to non_member, introduced above (see p. 25), except that instead of matching a single event-type template against a list of events, it matches a single event against a list of event-type templates; if both relations are interpreted as Prolog code, this distinction vanishes, since occurrences of "?" become unnamed Prolog variables, which match in either order by unification. In the following, this distinction is therefore ignored, and non_member is used in both cases.)

Finally, waiting_for_ack models the way Sender waits for an acknowledgment:

```
waiting_for_ack(Sender,Pkt,HE●[E])
    if   E = send(Sender,packet(Pkt,?),?,?).

waiting_for_ack(Sender,Pkt,HE●[E])
    if   non_member(E,[send(Sender,?,?,?)]),
         waiting_for_ack(Sender,Pkt,HE).
```

The first rule says that after Sender sends packet body Pkt, it is considered to be waiting for an acknowledgment for Pkt. The second rule says that if Sender is already waiting for an acknowledgment for Pkt after HE and if the next event E is not a send event, then Sender is still waiting for an acknowledgment for Pkt after event E has occurred.

A Hybrid System: A Railroad Crossing

One significant advantage of DMOD over many other temporal calculi is that it can be used to model hybrid systems, i.e., those whose state contains both discrete and continuous parameters. A convenient way of viewing hybrid systems is that they exist in phases. Each phase is modeled by well-behaved expressions in continuous mathematics, such as linear or differential equations. However, these expressions change when phase transitions occur, so that no single such expression describes behavior over the entire time line. Given the sequence of all phase transitions, it is possible to write down rules for recovering the phase that governs system behavior at any point in time. Assuming that such rules are easily expressed using definite clauses, we find that the crucial issue becomes computing the sequence of all phase transitions.

Phase transitions can be thought of as being caused by event occurrences. As discussed in Section 3, an event can denote an action, or it can denote the act of a proposition becoming true. For example, the phase transition that occurs when two billiards balls ball1 and ball2 collide at time T can be represented as the occurrence of the event distance_between(ball1,ball2,2*R,T). This denotes the proposition that the distance between the center of masses of ball1 and ball2 at time T is 2*R, where R is the (presumed identical) radius of each of the two balls. Such an event can be regarded as occurring at T when the proposition it denotes becomes true, i.e., it is true at T, but false immediately before T.

Since DMOD offers a way of specifying event occurrences, it can be employed for specifying how phase transitions occur. As noted in Section 3, DMOD's ability to model event preemption in hybrid systems is particularly useful. Rules for computing phases, given a history, can also be expressed using definite clauses, as shown above and elaborated below.

As an example of using DMOD to model phase transitions, we model a railroad crossing (Figure 2). This example is adapted from Ostroff [1989]; in particular, continuous time and position have been added. The model's discrete parameters are barrier and engine velocity, which change abruptly.

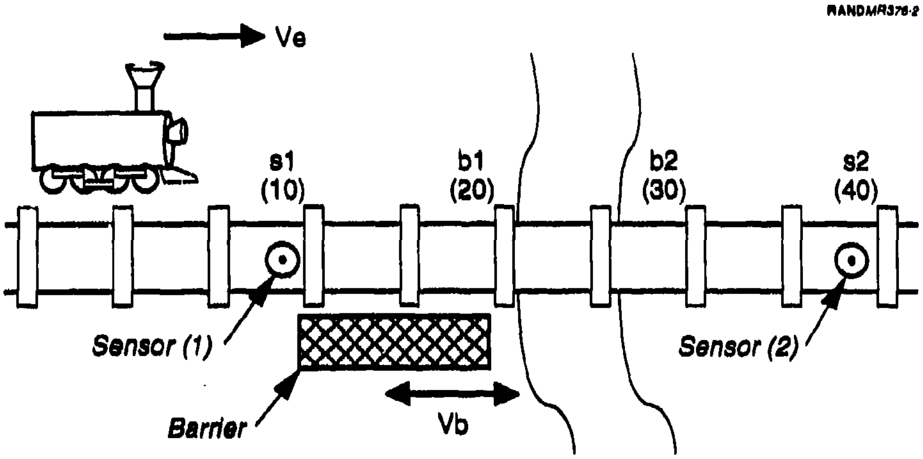

Figure 2—Railroad Crossing Model

In Figure 2, an engine moves on a track from left to right with a fixed velocity Ve. The track is crossed by a road between positions b1 and b2. A barrier slides back and forth to close or open the crossing. When an engine reaches position s1, sensor(1) detects the engine and starts to close the barrier. When an engine reaches position s2, sensor(2) detects the engine and starts to open the barrier. The barrier opens and closes at a finite speed Vb. Values for the positions of the sensors and the barrier are shown under their labels in parentheses (in arbitrary units).

Preemption arises in the system because closing or opening the barrier can be interrupted. For example, an engine arriving at s1 will start to close the barrier, but the engine may move so fast that before the barrier has fully closed, the engine arrives at s2 and starts to open the barrier again.

> **Notation convention:** We use time(H,T) to denote the relationship between a history H and the time stamp T of the last event in H (this can be thought of as a function that computes the time T of the last event in a history H).

Each event is of one of the following forms, with meanings given below:

> start(engine(X),Velocity,Position,T)
>
> sensed(engine(X),Sensor,T)
>
> start(barrier,close,T)
>
> end(barrier,close,T)

```
start(barrier,open,T)

end(barrier,open,T)

start(0).
```

The event start(engine(X),Velocity,Position,T) says that engine(X) starts to move with Velocity, from Position, at time T; sensed(engine(X),Sensor,T) says that engine(X) is sensed by Sensor at T; start(barrier,close,T) says that the barrier starts to close at T; end(barrier,close,T) says that the barrier finishes closing at T; start(barrier,open,T) and end(barrier,open,T) have the obvious analogous meanings; and start(0) represents the initial event in the system (with time stamp 0).

This model makes use of the relation travel_time, which can be defined as follows (using Prolog's arithmetic assignment operator "is") and which uses the simple relation absval to compute the absolute value of Speed:

```
travel_time(PrevPos,NewPos,Speed,TravelTime)
    if   absval(Speed,AbsSpeed),
         TravelTime is (NewPos – PrevPos) / AbsSpeed.

absval(X,X)
    if   X ≥ 0.

absval(X,–X)
    if   X < 0.
```

Given these definitions, the rules for the model (labeled rail-1 through rail-v10) are as follows:

```
rail-1        causal_connection(start(0),?,start(engine(1),ve,pe,t1),?).
```

The initial start(0) event causes the event of engine(1) starting to move (to the right), from a specific position pe, with a specific velocity ve, at time t1 (the last parameter, which represents the history between these two events, is a "don't care" in this case, written as "?").

```
rail-2a       causal_connection(E,HE,F,HEF)
                  if   E = start(engine(X),Velocity,Position,T),
                       F = sensed(engine(X),sensor(1),T+TravelTime),
                       travel_time(Position,s1,Velocity,TravelTime).
```

```
rail-2b     causal_connection(E,HE,F,HEF)
                if   E = start(engine(X),Velocity,Position,T),
                     F = sensed(engine(X),sensor(2),T+TravelTime),
                     travel_time(Position,s2,Velocity,TravelTime).
```

If engine(X) starts to move at time T with Velocity, from Position, then it is sensed by sensor(1) after the time taken to travel the distance between Position and s1 at Velocity. Its velocity always remains constant. The situation is similar for sensor(2).

```
rail-3      causal_connection(E,HE,F,HEF)
                if   E = sensed(engine(X),sensor(1),T),
                     F = start(barrier,close,T).
```

If engine(X) reaches sensor(1), then the barrier starts to close immediately.

```
rail-4      causal_connection(E,HE,F,HEF)
                if   E = sensed(engine(X),sensor(2),T),
                     F = start(barrier,open,T).
```

If engine(X) reaches sensor(2), then the barrier starts to open immediately.

```
rail-5      causal_connection(E,HE,F,HEF)
                if   E = start(barrier,close,T),
                     F = end(barrier,close,T+TravelTime),
                     position(barrier,Position,HE●[E]),
                     velocity(barrier,Velocity,HE●[E]),
                     travel_time(Position,b2,Velocity,TravelTime),
                     non_member(sensed(engine(?),sensor(2),?),HEF).
```

If the barrier starts to close, then it ends closing after the time needed for it to reach its fully closed position (b2) at its current velocity, provided no engine is sensed by sensor(2) before it finishes closing.

```
rail-6      causal_connection(E,HE,F,HEF)
                if   E = start(barrier,open,T),
                     F = end(barrier,open,T+TravelTime),
                     position(barrier,Position,HE●[E]),
                     velocity(barrier,Velocity,HE●[E]),
                     travel_time(Position,b1,Velocity,TravelTime),
                     non_member(sensed(engine(?),sensor(1),?),HEF).
```

If the barrier starts to open, then it ends opening after the time needed for it to reach its fully open position (b1) at its current velocity, provided no engine is sensed by sensor(1) before it finishes opening. (Note that although the sign of the velocity of the barrier reflects its direction of motion and the opening velocity is arbitrarily taken to be negative, the computation in travel_time uses absval to produce a positive value for TravelTime in all cases.)

The following "value" rules compute various state parameters, such as the position and velocity of engines or barriers. Note the use of the Prolog arithmetic assignment operator "is" and of the continuous functions +, - and * on the right hand side of this operator. More complex functions can similarly be used. The first value rule (rail-v1) specifies the value of position under continuous conditions:

```
rail-v1     position(Object,NewPos,HE•[E])
                if  position(Object,OldPos,HE),
                    velocity(Object,Velocity,HE),
                    time(E,TE),
                    time(HE,THE),
                    NewPos is OldPos + Velocity * (TE - THE).
```

This rule says that the position of any Object in the model (i.e., an engine or the barrier) is NewPos immediately after the last event E in a history HE•[E] if its position immediately after (the last event in) HE was OldPos, its velocity after HE was Velocity, and the change in its position is equal to the time between E and HE (which is TE - THE) multiplied by Velocity (which must have remained constant since no events can occur between HE and E). Since this rule computes position recursively, it must find an initial value from rules rail-v3 and rail-v5 below.

In addition to computing the position of an object after a given event, the model can also determine the position of an object at a certain time:

```
rail-v2     position_at(Object,T,NewPos,Hist_Thru_T)
                if  position(Object,OldPos,Hist_Thru_T),
                    velocity(Object,Velocity,Hist_Thru_T),
                    time(Hist_Thru_T,TH),
                    NewPos = OldPos + Velocity * (T - TH).
```

This rule is similar to the previous one, but it finds the position of any Object at a given time T, provided that Hist_Thru_T is the sequence of all events that have occurred whose time stamps are less than or equal to T. (The sequence Hist_Thru_T is easily constructed from any history by deleting all events after the first one whose time stamp is greater than T.)

Additional rules are required to specify the initial position and velocity of an engine after it starts since starting is a "phase transition" event that affects these values discontinuously:

rail-v3 position(engine(X),Position,H●[start(engine(X),Velocity, Position,T)]).

rail-v4 velocity(engine(X),Velocity,H●[start(engine(X),Velocity, Position,T)]).

The first (second) of these rules says that the position (velocity) of engine(X) immediately after an event of the form start(engine(X),Velocity,Position,T) is equal to the value of the Position (Velocity) parameter that appears in the start event itself.

Similar rules are needed for the position and velocity of the barrier after each of the phase transition events that affect these values discontinuously:

rail-v5 position(barrier,b1,[start(0)]).

rail-v6 velocity(barrier,0,[start(0)]).

Rules rail-v5 and rail-v6 specify the initial position (b1) and velocity (0) of the barrier.

rail-v7 velocity(barrier,-Vb,H●[start(barrier,open,?)]).

rail-v8 velocity(barrier,+Vb,H●[start(barrier,close,?)]).

rail-v9 velocity(barrier,0,H●[end(barrier,?,?)]).

Rules rail-v7 and rail-v8 specify the discontinuous change of velocity of the barrier immediately after it starts to open or close: These velocities have the same magnitude, Vb, but with opposite signs to reflect opposite directions of motion. Rule rail-v9 says that the velocity of the barrier becomes zero after any end event (whether it ends opening or ends closing).

Finally, if none of these phase transition events have just occurred, the velocity of the barrier remains unchanged from whatever its last phase transition made it:

```
rail-v10      velocity(barrier,Velocity,HE●[E])
                   if    non_member(E,[start(barrier,?,?),end(barrier,?,?)]),
                         velocity(barrier,Velocity,HE).
```

This rule says that the velocity of the barrier immediately after the last event E in a history H is the same as that immediately after the previous event in H (which is the last event in HE), if E is neither a start nor an end event involving the barrier.

5. Computing History from a DMOD Model

DMOD has been developed as an alternative to the discrete-event modeling technique to produce a formalism that allows reasoning about models. However, before showing how to reason about DMOD models, we must first show that DMOD provides at least as much functional power as the discrete-event modeling approach that it is intended to replace. In particular, we must show that DMOD can simulate the behavior of a system in something analogous to discrete-event simulation. In this section, we show two procedures for performing simulation with DMOD. The first of these is conceptually simple but relatively inefficient, while the second procedure is quite efficient.

The fundamental assumption (as discussed above in Section 3) implies that computing a history is the same as simulation. Therefore, to show how DMOD can perform simulation, we must show how to compute a history from a DMOD model. Intuitively, a history contains all—and only—those events whose occurrence is implied by the occurrence of the initial event and the causality rules. More formally:

> **Dfn:** If P is a DMOD program and $S = E_0, E_1, E_2, \ldots$ is a temporally ordered sequence of events in which any given event appears at most once, then S is *causally sound* (w.r.t. P) if every noninitial event in S has at least one cause in S, i.e., for any $j > 0$:
>
> $$\exists\, i \mid i < j \wedge \text{causal\_connection}(E_i, [E_0, \ldots, E_{i-1}], E_j, [E_{i+1}, \ldots, E_{j-1}])$$

Of course, a causally sound sequence may not contain all of the events that should intuitively occur (for example, the sequence $[E_0]$ is trivially causally sound from the definition, since E_j is not required to have a cause, for $j = 0$). For a sequence to satisfy the intuitive notion of a history, it must satisfy a causal-completeness property in addition:

> **Dfn:** If P is a DMOD program and $S = E_0, E_1, E_2, \ldots$ is a temporally ordered sequence of events in which any given event appears at most once, then S is said to be *causally complete* (w.r.t. P) if both of the following conditions are satisfied:
>
> (a) For any i, j, $0 \le i$, $0 \le j$, there is no event F such that:

(1) causal_connection(E_i,[$E_0, \ldots, E_{i-1}$],F,[$E_{i+1}, \ldots, E_j$]) succeeds, and

(2) [E_j,F,E_{j+1}] is temporally ordered, with F strictly earlier than E_{j+1}, and

(3) F does not occur in [$E_0, \ldots, E_j$].

(b) If S is finite (so that there is some k such that S = [$E_0, E_1, \ldots, E_k$]), then for any event G that does *not* occur in S, there is no i, $0 \leq i \leq k$ such that:

(1) [E_k,G] is temporally ordered, and

(2) causal_connection(E_i,[$E_0, \ldots, E_{i-1}$],G,[$E_{i+1}, \ldots, E_k$]) succeeds.

Informally, condition (a) states that no new event F can be caused after E_j but strictly before E_{j+1}, for any j [note that in (a1), the last event in HEF is E_j, which is therefore the event just prior to F]. Condition (b) says that S must not be extendable at the end, i.e., it must already contain all events that are caused by the events in S. Note that the sequence [E_0] always trivially satisfies (a) but not necessarily (b).

Finally, we can define a history, as follows:

Dfn: If P is a DMOD program and E_0 is a unique initial event, then a history is a sequence of events starting at E_0, which is both causally sound and causally complete.

Note that a single initial event is sufficient: Multiple initial events can always be preceded by a new, unique initial event that causes them all. Note also that this definition does not require that a history be finite.

A Simple Procedure for Computing History

For a DMOD program P with initial event E_0, we now develop a procedure for computing histories for P. This provides a simulation procedure for DMOD models.

Dfn: If S is a set of events, then E is the *earliest* event in S provided that for each event F in S, [E,F] is temporally ordered. Note that if S contains events concurrent with each other, there can be more than one earliest event in S.

Note 5.1 The effectiveness restriction on the ordering between time stamps discussed in Section 3 implies that a nonempty, finite set of events S has an earliest event that can be computed effectively.

Procedure 1: Enter E_0 as the initial event. Suppose the history $E_0, E_1, \ldots, E_m$, $m \geq 0$, has been computed so far. We need to compute the next event E_{m+1}. For each i, $0 \leq i \leq m$, let Effects(i) be the set of events:

$\{F_i \mid$ causal_connection(E_i,$[E_0, \ldots, E_{i-1}]$,F_i,$[E_{i+1}, \ldots, E_m]$) succeeds in the presence of P$\}$.

For any i, $0 \leq i \leq m$, let new_Effects(i) be the set of those events ε in Effects(i) such that:

(1) ε is not already in $[E_0, E_1, \ldots, E_m]$, and
(2) $[E_m, \varepsilon]$ is temporally ordered.

Let S_m be the union of new_Effects(i), $0 \leq i \leq m$. Take the next event E_{m+1} to be the earliest event in S_m. If S_m is empty, halt.

Intuitively, given a partial history H, this procedure determines the set S_m of all the events E, not in H, that are caused by an event E_i in H, in the temporally ordered context H•[E]. We pick as E_{m+1} the earliest event in S_m. Note that since S_m can contain more than one earliest event, this procedure is nondeterministic: A different history would be computed for each choice of E_{m+1}.

Theorem 5.1. Procedure 1 is effective.

Proof: Given $[E_0, E_1, \ldots, E_m]$, and E_i, $0 \leq i \leq m$, Effects(i) can be computed effectively. This follows directly from the DMOD effectiveness restriction (Section 3, p. 23). By the same restriction, Effects(i) is finite, so it is possible to effectively determine whether each event E in Effects(i) already occurs in $[E_0, \ldots, E_m]$. By the effectiveness restriction on the ordering between time stamps (Section 3, Linear Ordering Restriction 1, p. 19), it is also possible to effectively determine whether $[E_m, E]$ is temporally ordered. Therefore, new_Effects(i) can be computed effectively. The union of finite sets S_m can be formed effectively. By Note 5.1 above, if S_m is nonempty, an earliest event in S_m exists and can be computed effectively.

Theorem 5.2. Soundness and completeness of Procedure 1: If P is a DMOD program and E_0 is an initial event, then a sequence of events $[E_0, E_1, \ldots]$ is computed by Procedure 1 if and only if it is a history.

Proof: by contradiction, in two parts ("if" and "only if"):

"If" part: Suppose $[E_0, E_1, \ldots]$ has been computed. Then, by the statement of the procedure above, an event appears at most once in the sequence. Furthermore, the sequence is temporally ordered.

Suppose that the sequence is not causally sound. Since $[E_0]$ is causally sound, there exists some k, $0 < k$ such that $[E_0, E_1, \ldots, E_{k-1}]$ is causally sound, but $[E_0, E_1, \ldots, E_{k-1}, E_k]$ is not. Then there is no j, $0 \leq j \leq k-1$ such that causal_connection(E_j, $[E_0, \ldots, E_{j-1}]$,E_k,$[E_{j+1}, \ldots, E_{k-1}]$) succeeds. Since E_k has been computed as the next event after E_{k-1}, it must belong to the set S_{k-1} in the procedure. But by the definition of S_{k-1}, there must exist an E_j as above. (Contradiction.)

Suppose that the sequence is not causally complete. Then either condition (a) or (b) of the definition of causal completeness (see above) must be violated. Suppose condition (a) is violated. Then there exist i,j, $0 \leq i$, $0 \leq j$, and event F such that:

(i) causal_connection(E_i,[E_0, . . . , E_{i-1}],F,[E_{i+1}, . . . , E_j]), succeeds

(ii) [E_j,F,E_{j+1}] is temporally ordered, with F strictly earlier than E_{j+1}, and

(iii) F does not occur in [E_0,E_1, . . . , E_{j+1}].

Then F belongs to the set S_j in the procedure. Since E_{j+1} is computed as the next event after E_j, it also belongs to S_j and is an earliest event in it. Therefore, [E_{j+1},F] must be temporally ordered. However, by (ii) above and the trichotomy restriction on the ordering between time stamps (Section 3, Linear Ordering Restriction 2), [E_{j+1},F] cannot be temporally ordered. (Contradiction.)

Suppose condition (b) is violated. Then, there exists some k such that the computed sequence is [E_0,E_1, . . . , E_k]. Also, there exists an event G not occurring in [E_0,E_1, . . . , E_k], and i, $0 \leq i \leq k$, such that [E_k,G] is temporally ordered, and causal_connection(E_i,[E_0, . . . , E_{i-1}],G,[E_{i+1}, . . . , E_k]) succeeds. Then, the set S_k is not empty. Therefore, the termination condition of Procedure 1 is violated. (Contradiction.)

"Only if" part: Suppose [E_0,E_1, . . .] is a history that is not computed by Procedure 1. Since E_0 is computed, there exists some j, $0 < j$ such that [E_0,E_1, . . . , E_{j-1}] is computed but E_j is not. But since [E_0,E_1, . . . , E_j] is causally sound, there must also exist some i, $0 \leq i \leq j - 1$ such that causal_connection(E_i,[E_0, . . . , E_{i-1}],E_j,[E_{i+1}, . . . , E_{j-1}]) succeeds. Since [E_0,E_1, . . .] is a history, no event can occur more than once in it, so E_j cannot occur in [E_0,E_1, . . . , E_{j-1}]. In addition, [E_{j-1},E_j] is temporally ordered, so E_j belongs to the set S_{j-1} in the procedure.

Since S_j is nonempty and E_j is not computed (by assumption), there must be another event F in S_{j-1} such that F is strictly earlier than E_j. Since F belongs to S_{j-1}, there must exist some m, $0 \leq m < j$ such that:

(iv) causal_connection(E_m,[E_0, . . . ,E_{m-1}],F,[E_{m+1}, . . . , E_{j-1}]) succeeds,

(v) $[E_{j-1},F,E_j]$ is temporally ordered, with F strictly earlier than E_j, and

(vi) F does not occur in $[E_0, \ldots, E_{j-1}]$.

However, this contradicts condition (a) for causal completeness of the history $E_0,E_1, \ldots$ (Section 5).

Simulating the Railroad Crossing

We illustrate the above simulation procedure using the rail example from Section 4. We assume a velocity for the barrier of Vb = 10 (which was unspecified in Section 4).

Let E_0 = start(0) occur as the initial event. Rule rail-1 matches E_0, producing the set:

Σ_0 = { start(engine(1),40,0,0) }

where the initial velocity of the engine (ve) is 40 and its initial position (pe) is 0. The next event is therefore:

E_1= start(engine(1),40,0,0).

Now rules rail-2a and rail-2b each match E_1, producing two sensed events. The first of these, in which sensor(1) senses engine(1), will occur after engine(1) has traveled from its starting position (which is 0) to the position s1 of sensor(1) at the engine's velocity (ve): It will, therefore, occur at time T = (s1 – 0)/ve = 10/40 = 0.25. The second sensed event will occur when engine(1) reaches the position s2 of sensor(2), which will occur at T = (s2 – 0)/ve = 40/40 = 1.0, producing:

Σ_1 = { sensed(engine(1),sensor(1),0.25), sensed(engine(1),sensor(2),1.0) }.

Note that although causal_connection(E_0,[],E_1,[E_1]) succeeds by rule rail-1, E_1 is not a member of Σ_1 since it has already occurred. Therefore, the next event is:

E_2 = sensed(engine(1),sensor(1),0.25).

Proceeding in this way, we produce the following history (with comments in italics):

E0 = start(0)
E1 = start(engine(1),40,0,0)
E2 = sensed(engine(1),sensor(1),0.25) *(0.25 = (s1 – 0)/40)*
E3 = start(barrier,close,0.25)
E4 = sensed(engine(1),sensor(2),1.0) *(1.0 = (s2 – 0)/40)*
E5 = start(barrier,open,1.0)
E6 = end(barrier,open,1.75).

Note that at time 0.25, the barrier starts to close at velocity Vb = 10 (as specified above); this should cause it to end closing at time 1.25, but the engine reaches the position of the second sensor (s2) at time 1.0 before the barrier has fully closed. Therefore, even though the event G = end(barrier,close,1.25) is potentially caused by E_3, its occurrence is preempted by E_4 = sensed(engine(1),sensor(2),1.0). That is, causal_connection(E_3,[E_0,E_1,E_2],G,[E_4]) does not succeed, since the presence of E_4 makes the last condition in the body of rule rail-5 false. Therefore G is not in Σ_4 (or Σ_5 or Σ_6). Even though G is in Σ_3, it does not appear in the history because E_4 also appears in Σ_3, and since the time stamp of E_4 is earlier than that of G, it preempts G.

Note that this algorithm makes no use of devices such as event queues or scheduling and unscheduling, which are the basis of the event-scheduling view of the discrete-event technique [Evans, 1988]. The resulting simplification makes reasoning about models considerably easier.

An Improved Procedure to Compute History

The above procedure is simple but quite inefficient. To compute the set Σ_m, we need to determine for each E_i whether there exists F_i such that causal_connection(E_i,HE_i,F_i,HE_iF_i) succeeds. After E_{m+1} has been computed we repeat this computation for all i, except i = m + 1, to determine Σ_{m+1}. In other words, we do not incrementally derive Σ_{m+1} from Σ_m.

This section develops an improved procedure that avoids such recomputation. It maintains a queue of items, each of which consists of a potentially caused event, E, with an associated "guard" condition: this guard condition applies to the time period starting at the causing event and ending at the caused event (E). It is evaluated when the history up to the time stamp of E has been computed. If the guard condition is true, the event E is recorded in the history, otherwise it is discarded. This scheme allows the queue for the next cycle of this procedure to be computed incrementally from the current queue.

This procedure is similar in spirit to the traditional discrete-event simulation procedure, which schedules and unschedules events on an event queue. However, there are two fundamental differences. First, DMOD programs do not manipulate the event queue at all, and (more generally) the simulation procedure is invisible to the programmer, which eliminates large classes of potential bugs (including explicit mismanagement of the event queue). Second, in the discrete-event procedure, an event is unscheduled as soon as it is determined that it cannot occur. In DMOD, unscheduling occurs only when the history up to the time stamp on the event has been accumulated and the associated condition is found to be false. The difference is between "looking forward" in the traditional discrete-event procedure versus "looking backward" in DMOD, which is far simpler.

The procedure is restricted to DMOD programs in which each causality rule can be expressed in the form:

```
causal_connection(E,HE,F,HEF)
    if    connection_predicted(E,HE,F),
          prediction_unfalsified(E,HE,F,HEF).
```

This restriction does not lead to much loss of expressiveness. The rule states that if, from the information up to E, a causal connection between E and F is predicted and this prediction is unfalsified by information in HEF, then E is causally connected to F in the context HE●[E]●HEF●[F]. This effectively divides the body of a causal_connection rule into two parts, one to be evaluated from the past of E and the other from the future of E.

Let P be a DMOD program in which each causal_connection rule can be expressed in the above form. Then the following are both true:

(1) For any ground terms E and HE, the set of all ground terms F such that connection_predicted(E,HE,F) succeeds is finite and can be computed in finite time.

(2) For any ground terms E,HE,F,HEF, it is always possible to determine, in finite time, whether prediction_unfalsified(E,HE,F,HEF) succeeds.

Using this approach requires reformulating the rules in the DMOD models above. For example, rule rail-6 from the railroad crossing model above:

```
rail-6        causal_connection(E,HE,F,HEF)
                  if   E = start(barrier,open,T),
                       F = end(barrier,open,T+TravelTime),
                       position(barrier,Position,HE●[E]),
                       velocity(barrier,Velocity,HE●[E]),
                       travel_time(Position,b1,Velocity,TravelTime),
                       non_member(sensed(engine(?),sensor(1),?),HEF).
```

can be reformulated as:

```
              causal_connection(E,HE,F,HEF)
                  if   connection_predicted(E,HE,F),
                       prediction_unfalsified(E,HE,F,HEF).

              connection_predicted(E,HE,F)
                  if   E = start(barrier,open,T),
                       F = end(barrier,open,T+TravelTime),
                       position(barrier,Position,HE●[E]),
                       velocity(barrier,Velocity,HE●[E]),
                       travel_time(Position,b1,Velocity,TravelTime).

              prediction_unfalsified(E,HE,F,HEF)
                  if   E = start(barrier,open,T),
                       F = end(barrier,open,T+TravelTime),
                       non_member(sensed(engine(?),sensor(1),?),HEF).
```

We also assume that every DMOD program contains the "default" rule:

```
default-rule  causal_connection(E,HE,F,HEF)
                  if   connection_predicted(E,HE,F),
                       prediction_unfalsified(E,HE,F,HEF).
```

This rule is needed because in the procedure below we would like to be able to infer that for *every* E,HE,F,HEF, if connection_predicted (E,HE,F) and prediction_unfalcified(E,HE,F,HEF) both succeed, then causal_connection (E,HE,F,HEF) succeeds. Without this default rule, we cannot always infer this, even if all rules for causal_connection are of this form.

Notation convention: We use the lambda-notation for representing functions. An expression of the form λx.E, where x is a variable and E is an expression, denotes a function of one argument. The result of applying it to a ground expression arg, is the expression obtained by replacing x by arg in E. For example, $\lambda x.x > 0$ denotes the function "is a positive number." The result of applying it to 1 is $1 > 0$, which yields the value *true*.

Procedure 2: For a DMOD program P with initial event E_0, first enter E_0 in the history. Let $[E_0, \ldots, E_m]$ be the history computed up to some point of time. Let Q_m be a set of functions of the form λHEF.prediction_unfalsified(E,HE,F,HEF) where E,HE,F are all ground, but HEF is a variable. An item λHEF.prediction_unfalsified(E,HE,F,HEF) is in Q_m if and only if there exists i, $0 \leq i \leq m$ such that:

(a) $E = E_i$ and $HE = E_0, E_1, \ldots, E_{i-1}$, and
(b) connection_predicted(E,HE,F) succeeds, and
(c) $[E_m, F]$ is temporally ordered, and
(d) F does not appear in $E_0, \ldots, E_m$.

Let R_m be the set of all events F such that λHEF.prediction_unfalsified (E,HE,F,HEF) is in Qm and succeeds with $HEF = [E_{i+1}, \ldots, E_m]$, where for some i, $0 \leq i \leq m$, $E = E_i$, and $HE = [E_0, \ldots, E_{i-1}]$. Take E_{m+1} to be an earliest event in R_m. If R_m is empty, the algorithm halts.

To compute Q_{m+1}, delete from Q_m the function from which E_{m+1} was derived, as well as all functions λHEF.prediction_unfalsified (E,HE,F,HEF) such that $[E_{m+1}, F]$ is not temporally ordered. Then add to Q_m all items λHEF.prediction_unfalsified(E_{m+1},HE_{m+1},F_{m+1},HEF) where $HE_{m+1} = [E_0, \ldots, E_m]$, such that:

(i) connection_predicted(E_{m+1},HE_{m+1},F_{m+1}) succeeds, and
(ii) $[E_{m+1}, F_{m+1}]$ is temporally ordered, and
(iii) F_{m+1} does not appear in $E_0, E_1, \ldots, E_{m+1}$.

Note that connection_predicted is evaluated only for E_{m+1}. All functions contributed by $E_0, \ldots, E_m$ are already in Q_{m+1}, if not deleted above. Condition (d) is enforced in two steps. First, the function from which E_{m+1} is derived is deleted. Second, no function generated by E_{m+1} in which the caused event already occurs in $E_0, \ldots, E_{m+1}$ is included in Q_{m+1}. Therefore, Q_{m+1} satisfies conditions (a) through (d) above. Moreover, it is derived incrementally from Q_m as we had desired.

The effectiveness of Procedure 2 can be proved in the same way as for Procedure 1. We now state and prove its correctness.

Theorem 5.3. Soundness and completeness of improved procedure. If P is a DMOD program and E_0 is an initial event, then a sequence of events $[E_0, E_1, \ldots]$ is computed by Procedure 2 if and only if it is a history.

Proof: Suppose $[E_0, \ldots, E_m]$, $m \geq 0$ has been computed. We show that the set Σ_m of Procedure 1 above is identical to the set R_m of Procedure 2. In both cases, an earliest next event is computed as E_{m+1}.

Let F_j belong to R_m. Then by condition (d) above, F_j is not equal to any E_i, $0 \leq i \leq m$, and there exist i, $i \leq m$ and λHEF.prediction_unfalsified $(E_i,[E_0, \ldots, E_{i-1}],F_j,\text{HEF})$ in Q_m such that prediction_unfalsified $(E_i,[E_0, \ldots, E_{i-1}],F_j,[E_{i+1}, \ldots, E_m])$ succeeds. The membership of this function in Q_m implies that connection_predicted(E_i,HE_i,F_j) also succeeds, and that $[E_m,F_j]$ is temporally ordered. Therefore, by the default-rule for causal_connection (defined above, p. 43), causal_connection$(E_i,HE_i,F_j,\text{HEF})$ also succeeds, so F_j belongs to Σ_m.

Let F_j belong to Σ_m. Then it is not equal to any E_i, $0 \leq i \leq m$. Also $[E_m,F_j]$ is temporally ordered. Finally, there exist E_i, HE_i such that causal_connection$(E_i,HE_i,F_j,\text{HEF})$ succeeds where $HE_i = [E_0, \ldots, E_{i-1}]$ and HEF $= [E_{i+1}, \ldots, E_m]$. Then connection_predicted (E_i,HE_i,F_j) succeeds, by the definition of causal_connection. Therefore, the function λH.prediction_unfalsified(E_i,HE_i,F_j,H) is in Q_m. Again, by definition of causal_connection, prediction_unfalsified$(E_i,HE_i,F_j,[E_{i+1}, \ldots, E_m])$ succeeds. Therefore, F_j belongs to R_m.

6. Reasoning About DMOD Models

One of the main motivations for developing DMOD was to be able to reason about models developed using the discrete-event technique. In this section we present the general framework in which propositions about DMOD programs can be expressed and proved. Although such propositions are in general undecidable, we present two heuristics for guiding the search for proofs.

Properties of DMOD programs are expressed and proved at the "metalevel" of DMOD, as will become apparent below. While one could prove arbitrary properties about such programs, a natural class of properties is about their histories. Let P be a DMOD program, and E_0 a special initial event. Let history(X) be an abbreviation for the conjunction of the following conditions:

(a) X is a finite, or infinite, sequence of events with E_0 as the first event
(b) X is temporally ordered
(c) X is causally sound
(d) X is causally complete.

Let r be a condition on sequences of events. Examples of r are "safety" and "liveness" properties: Proving a safety property involves proving that some undesirable condition can never occur in a system, whereas proving a liveness property involves proving that some desirable condition will eventually occur [Lamport, 1983].

To show that *every* history for P satisfies r, we would prove the metalevel proposition:

$$\forall X \; [\; \text{history}(X) \supset r(X) \;]$$

whereas to show that *some* history satisfies *r*, we would prove:

$$\exists \, X \mid \text{history}(X) \wedge r(X).$$

Despite the effectiveness restrictions on DMOD programs, such propositions are, in general, undecidable. It is easy to model "terms" as events, "rewrite rules" [O'Donnell, 1985] as causality rules, and "rewriting" as computation of histories. Rewrite rules can model a Turing Machine, but the halting problem places limits on what can be proved in this way.

The ease with which such propositions can be proved depends, of course, on theorem provers for the metalanguage of P in which these propositions are formulated. However, we now show how the intuitive nature of causality and the logic of definite clauses and SLD-resolution can be used to greatly simplify the search for proofs of temporal properties. The fundamentals of SLD-resolution can be found in Hill [1974] or in Lloyd [1984].

A Heuristic for Proving Properties

We now present a heuristic for proving safety properties, as discussed above. This heuristic is based on the idea that causality provides a convenient way of answering two basic questions: whether an event occurs and whether an event does not occur.

Heuristic: To show that an event occurs, we need show only that at least one of its causes occurs; to show that a noninitial event does not occur, we need show only that none of its causes occur (or, alternatively, that none of the events that actually occur are its causes).

Since we have defined causality using definite clauses, these questions are reduced to questions about the existence or nonexistence of successful SLD-derivations. Such proofs have a simple structure, making it straightforward to answer many such questions.

If temporal properties can be expressed in terms of questions about event occurrences, then the above heuristic can be utilized. For example, one safety property in the context of the railroad crossing model (above) might be "For every T and X, if start(barrier,close,T) occurs then end(barrier,close,X) occurs, such that X – T is less than some critical amount." Similarly, the property that the barrier velocity eventually becomes positive can be expressed as "For some X, the event start(barrier,close,X) occurs."

We now formalize this heuristic for our special view of causality, causal_connection, by means of the following theorems at the metalevel of DMOD programs.

Theorem 6.1. Every noninitial event has a cause. If P is a DMOD program and $E_0, E_1, \ldots$ is a history computed from P, then:

$$\forall k \, [\, k > 0 \supset \exists i \mid i < k \wedge \text{causal\_connection}(E_i, [E_0, E_1, \ldots, E_{i-1}], E_k, [E_{i+1}, \ldots, E_{k-1}]) \text{ succeeds}].$$

Proof: Direct, from causal soundness of history.

Theorem 6.2. If an event does not occur, then it must not have a cause. If P is a DMOD program and $H = E_0, E_1, \ldots$ is a history computed from P, F is an event not occurring in H, and E_k is the last event in H such that $[E_k, F]$ is temporally ordered. Then:

$$\neg\exists\, i \mid 0 \leq i \leq k \wedge \text{causal\_connection}(E_i, [E_0, E_1, \ldots, E_{i-1}], F, [E_{i+1}, \ldots, E_k])$$
$$\text{succeeds.}$$

Proof: Direct, from causal-completeness of history.

To show that an event F does not occur in a history, assume that it does. Then, by Theorem 6.1, there must be an event E prior to F in the history such that it is possible to show, by SLD-resolution, that E is causally connected to F. Again, Theorem 6.1 can be applied to infer the existence of an event G, prior to E, such that it is possible to show by SLD-resolution that G is causally connected to E. In this way, generate a "backward causality chain" until a contradiction is derived.

Second, to show that an event F occurs in a history, assume it does not. Then, by Theorem 6.2, there can be no event E strictly earlier than F in the history, for which it is possible to show (by SLD-resolution) that E is causally connected to F. Show that at least one such event exists, thereby deriving a contradiction.

Example 1

In the railroad crossing example of Section 5, suppose the engine were to send a signal to a satellite every microsecond reporting its position. This could be represented by the following causality rules:

```
causal_connection(start(0),?,tell_satellite(engine(1),Position,0),?)
    if    position(engine(1),Position,[start(0)]).

causal_connection(E,HE,F,HEF)
    if    E = tell_satellite(engine(X),?,T),
          F = tell_satellite(engine(X),Position,T+Delay),
          position(engine(X),Position,HE•[E]•HEF•[F]),
          loop_delay(Delay).

loop_delay(10^ - 6).
```

The first rule says that the initial event causes engine(1) to tell the satellite its position immediately after the initial event. The second rule says that when engine(X) tells its position at T, it tells its position again at T+Delay. The third clause fixes Delay to be 1 microsecond.

Now, suppose we wanted to show that the event sensed(engine(1),sensor(2),1.0) occurs in each history of the system. We could, of course, generate all histories of the system up to time 1.0 and check whether this event occurs, but every such history would contain a million events of the engine transmitting its position to the satellite, and each of these events is irrelevant to the sensing of engine(1) by sensor(2).

We now show how the occurrence of this event can be proved, quite succinctly, using the above heuristic. Basically, we show that sensing is caused by an engine starting to move. In turn this is caused by the initial event. No consideration is given to communication events.

Notation convention. When we say an event E occurs, we mean that E occurs in every history of the model with initial event start(0). (The model here consists of the DMOD program for the railroad crossing presented in Section 4, augmented with the above three clauses to represent engine-to-satellite communication.)

Lemma 6.1. The event E = start(engine(1),40,0,0) occurs.

Proof: Suppose E does not occur. Let E_0 = start(0), H = $[E_0, \ldots, E_m]$ be a history for P, and E_k the last event in H such that $[E_k,E]$ is temporally ordered. By Theorem 6.2, there does not exist i, i < k such that causal_connection(E_i,$[E_0,E_1, \ldots, E_{i-1}]$,E,$[E_{i+1}, \ldots, E_k]$) succeeds. However, by rule rail-1, we have the successful SLD-derivation:

{causal_connection(start(0),[],start(engine(1),40,0,0),[e2, . . . , ek])}.

(Contradiction.)

Proposition 6.1. The event E = sensed(engine(1),sensor(2),1.0) occurs.

Proof: Suppose E does not occur. Let E_0 = start(0), H = $[E_0, \ldots, E_m]$ be a history for P, and E_k the last event in H such that $[E_k,E]$ is temporally ordered. By Theorem 6.2, there does not exist i, i < k such that causal_connection(E_i,$[E_0,E_1, \ldots, E_{i-1}]$,E,$[E_{i+1}, \ldots, E_k]$) succeeds.

By Lemma 6.1, we infer that start(engine(1),40,0,0) occurs. Let this be E_i.

Since E_i is strictly earlier than E, there is no successful SLD-derivation starting at causal_connection (E_i,[E_0,E_1, . . . , E_{i-1}], E,[E_{i+1}, . . . , E_k]). However, this matches rule rail-2b to yield the successful SLD-derivation:

{start(engine(1),40,0,0) = start(engine(X),Velocity,Position,T),
sensed(engine(1),sensor(2),1.0) =
sensed(engine(1),sensor(2),T+TravelTime),
TravelTime = (40 – Position)/Velocity}

{sensed(engine(1),sensor(2),1.0) = sensed(engine(1),sensor(2),TravelTime),
TravelTime = (40 – 0)/40}

{1.0 = (40 – 0)/40}

(Contradiction.)

This proves the desired result, i.e., that the event sensed(engine(1),sensor(2),1.0) occurs.

Example 2

We now prove that the event end(barrier,close,1.25) does not occur. This is somewhat trickier than the previous example since it involves event preemption.

Lemma 6.2. The event sensed(engine(1),sensor(1),0.25) occurs.

Proof: Identical to that of Lemma 6.1, but using rule rail-2a.

Lemma 6.3. The event start(barrier,close,0.25) occurs.

Proof: Similar to that of Lemma 6.1 but using rule rail-3.

Lemma 6.4. If the event E = start(engine(1),40,0,X) occurs then X = 0.

Proof: Suppose E occurs. If E_0 = start(0) and H = [E_0, . . . , E_m] is a history for P, let E = E_k. Then, by Theorem 6.1, there must exist some i, $0 \leq i < k$ such that causal_connection(E_i,[E_0, . . . , E_{i-1}],E_k,[E_{i+1}, . . . , E_{k-1}]) succeeds. Because of the form of E_k and of rules in P, the only successful derivation can be obtained from rule rail-1. There is one such successful derivation, which yields X = 0.

Lemma 6.5. If the event E = start(barrier,close,X) occurs then X = 0.25.

Proof: Analogous to Lemma 6.4: Using rule rail-2a, first show that if sensed(engine(1),sensor(1),X) occurs then X = 0.25. Then using rule rail-3 similarly proves the lemma.

Proposition 6.2. The event E = end(barrier,close,1.25) does not occur.

Proof: Suppose E occurs. Let E_0 = start(0), H = [$E_0, \ldots, E_m$] be a history for P and let E = E_k. Then by Theorem 6.1, there must exist some i, $0 \leq i < k$ such that causal_connection(E_i,[$E_0, \ldots, E_{i-1}$],E_k,[$E_{i+1}, \ldots, E_{k-1}$]) succeeds.

Because of the form of E_k and of rules in P, the only successful derivation can be obtained from rule rail-5. Therefore, there must be a successful SLD-derivation starting at:

{E_i = start(barrier,close,T),
end(barrier,close,1.25) = end(barrier,close,T+TravelTime),
position(barrier,Position,[$E_0, \ldots, E_i$]),
velocity(barrier,Velocity,[$E_0, \ldots, E_i$]),
TravelTime = (b2 – Position)/Velocity,
non_member(sensed(engine(?),sensor(2),?),HEF)}

By Lemma 6.5, T = 0.25. HEF is a sequence of events in the history whose first event has time stamp greater than or equal to 0.25 and whose last event has time stamp less than or equal to 1.25. By Proposition 6.1 sensed(engine(1),sensor(2),1.0) is in the history. Since $0.25 \leq 1.0 \leq 1.25$, this event is in HEF. However, now, the last condition in the goal cannot succeed, producing a contradiction that proves Proposition 6.2.

7. Relationship with Previous Work

Event Scheduling View of the Discrete-Event Technique

DMOD was the result of an attempt to provide a logical basis for the widely used event-scheduling view of the discrete-event modeling technique [Fishman, 1973; Zeigler, 1984; Evans, 1988; IEEE, 1989]. Without such a basis, it is quite difficult to reason about discrete-event models.

The event-scheduling approach to discrete-event modeling maintains a clock and a central queue of time-stamped events. When an event occurs, the clock time is advanced to be the time stamp on this event, and the system state is updated to correspond to this time. All events that the occurring event can possibly cause are then scheduled (inserted) into an event queue. All events in the queue that the occurring event precludes are unscheduled (deleted) from the event queue. The next occurring event is taken to be the one with the earliest time stamp. A discrete-event model specifies the scheduling and unscheduling relationships between events as well as how state is updated when events occur.

The logical meaning of scheduling and unscheduling is, however, far from obvious. "E schedules F" cannot be taken to mean "E causes F." If this were so, then if E occurred, so would F. There would be no possibility of unscheduling F. Therefore, "E schedules F" and "G unschedules F" can be taken to mean "E causes F provided G does not occur in between." The attempt to formalize this idea led us to the difficulties discussed in Section 3, whereas the attempt to solve these difficulties led to the concept of causal connection presented above.

Other Work

Many temporal formalisms view systems as transitioning from one discrete state to another. This view can be inappropriate when continuous parameters are involved. For example, it is impossible to write a state-transition function defining the next position of a moving object. One approach would be to regard the phases of a hybrid system as higher-level states, and thereby still regard these as state-transition systems. But if it is natural to regard continuous parameters as part of the state, then this would require distinguishing between parameters for which state-transition functions can and cannot be written. This would be quite awkward in practice.

DMOD adopts the "event-occurrence" view of systems. It assumes that the behavior of a system can be recovered from its history. It provides a way of expressing how events occur by means of causality and definite clauses. Perhaps the most important feature of DMOD is its ability to model—in a logical but computationally tractable manner—the way the effects of events depend on their future, in continuous time. A special case of this phenomenon is event preemption, which has been discussed above: It is useful for modeling hybrid as well as discrete systems.

There appear to be few systems in which event preemption is modeled in a formal way. One example is Petri Nets [Peterson, 1977] with inhibitor arcs. DMOD is, perhaps, more expressive due to its basis in definite clauses. Another example is the L.0 language [Cameron et al., 1990; Ness, 1990], in which cause-effect rules using the "until" operator can be utilized. However, unlike DMOD, only discrete time is modeled in L.0. This allows the L.0 interpreter to iterate over each point of time in an interval to check whether a preemption condition is satisfied in it. As discussed above, this approach does not work when time is continuous.

Temporal Logic [Pnueli, 1977; Manna & Pnueli, 1981] is derived from full first-order logic by disallowing explicit mention of time. This simplifies the logic but keeps it expressive enough for studying a wide range of systems, such as non-real-time concurrent programs. Of course, it is not possible to use it to model systems such as those we considered, in which precise timing information needs to be expressed. Attempts to introduce explicit time into Temporal Logic are described in Ostroff [1989] and Alur et al. [1990]. The first two of these attempts are intended only for discrete time, so they cannot be used for hybrid systems; the last is interpreted over continuous time, but it is still restricted to specifying only fixed temporal bounds.

DMOD suggests an alternative way to avoid the use of full first-order logic to model temporal knowledge. It shows how to do this using a relatively tractable subset of full first-order logic, namely, the logic of definite clauses.

The event occurrence view is also taken by Inan & Varaiya [1987]. Processes are described by means of recursion equations, and an algebra of processes is presented. However, event preemption does not seem to be discussed. Sandewall [1989] models hybrid systems using full first-order logic, and McDermott [1982] presents a logic for reasoning about hybrid systems. Many axioms about causality are presented in the literature, but there is not much discussion of the computational issue of how causality is to be inferred from these axioms. DMOD has attempted to resolve these issues.

Kowalski & Sergot [1986] and Kowalski [1986] have proposed a definite clause-based event calculus for the purpose of assimilating narratives. Given an account of what events occurred, it can be employed to determine what relationships hold over what periods of time. Therefore, their ideas can be employed for analyzing DMOD histories. However, the calculus does not discuss event preemption. The logic of Allen [1984] also does not discuss such a mechanism.

Qualitative Reasoning [Kuipers, 1986; de Kleer & Brown, 1984; Forbus, 1984] proposes qualitative discretization of real-valued spaces over which continuous parameters normally range. For example, temperature may range over the set {low, medium, high}. It further proposes reconstruction of tools of traditional mathematics, such as differential equations for discrete spaces. However, by its very nature, it does not seem appropriate when exact predictions need to be made.

8. Current Status and Future Directions

As discussed above, the DMOD formalism is an alternative to the object-oriented simulation approach pioneered at RAND and elsewhere. DMOD facilitates building models that answer questions beyond "*What if . . . ?*" The feasibility of this approach was initially demonstrated by building a logistics model that reproduced the important features of an existing object-oriented simulation of a corps-sized distribution environment (the Wartime Theater Ammunition Distribution System, WTADS [Schank et al., 1990]).

The DMOD formalism has subsequently been refined and extended in a number of ways and has been used to produce an initial prototype strategic mobility model (SMM) with a number of novel properties. This initial version of SMM consists of a basic model of the physical aspects of a strategic mobility environment plus an embryonic model of management policies. This model is designed as an integrated simulation and planning model that facilitates the use of simulation in plan evaluation, construction, and modification and supports a number of "Beyond *What if . . . ?*" capabilities. All aspects of the model are defined declaratively in terms of causal relationships among events and the objects that participate in those events. The management policy submodel, while relatively minimal, is appropriately encapsulated to distinguish between physical causes and causes resulting from human decisionmaking.

The DMOD models implemented to date allow tracing causal chains, both forward and backward, both in a concrete history (i.e., finding which events *actually* caused which others in a given simulation run) and (with some restrictions) in the abstract (i.e., finding which events *can* cause which others in a model). DMOD also provides a novel analytic capability for answering unanticipated (ad hoc) queries without reinstrumenting or rerunning a simulation. That is, questions can be asked about state parameters that were not even defined when a simulation was run; such questions can be answered on the order of 60 times faster than rerunning the simulation after reinstrumenting it. In addition, DMOD provides a novel capability for direct, interactive validation, in which the user witnesses causal relationships and their effects directly as a model runs, rather than having to infer them from the model's behavior. This helps domain experts and analysts understand the behavior of the model without having to understand its implementation.

A number of heuristics have been developed for reasoning about DMOD models, as required to answer questions "Beyond *What if. . . ?*" The most promising of these have been implemented in the evolving SMM prototype to produce and analyze abstract causal chains and plans; the SMM prototype is capable of producing simple, partially instantiated plans. This work is open-ended and ongoing; it promises to be highly relevant to the kinds of long-range questions asked by strategic mobility planners. In conjunction with other projects at RAND, the RASL project has brought this work to the attention of the Logistics Directorate of the Joint Staff, which has expressed considerable interest in the potential use of these techniques in studies such as the Mobility Requirements Study.

References

Allen, J. F. [1984]. Towards a general theory of action and time. *Artificial Intelligence*, Vol. 23, No. 2, pp. 123–154.

Alur, R., Courcoubetis, C., Dill, D. [1990]. Model checking for real-time systems. *Proceedings of Fifth IEEE Annual Symposium on Logic in Computer Science.* Philadelphia, PA.

Apt, K. R., van Emden, M. H. [1982]. Contributions to the theory of logic programming. *Journal of the ACM*, Vol. 29, No. 3.

Cameron, E., Cohen, D., Ness, L., Srinidhi, H. [1990]. L.0: A language for modeling and prototyping communications software. *Third International Conference on Formal Description Techniques, (FORTE)*, Madrid.

Cammarata, S., Gates, B., Rothenberg, J. [1988]. *Dependencies, Demons and Graphical Interfaces in the ROSS Language.* N-2589-DARPA, RAND, Santa Monica, CA.

Carolan, W. J., Hill, J. E., Kennington, J. L., Niemi, S., and Wichmann, S. J. [1990]. An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research*, Vol. 38, No. 2, March–April, pp. 240–248.

Davis, E. [1988]. A logical framework for commonsense predictions of solid object behavior. *Artificial Intelligence in Engineering*, Vol. 3, No. 3.

Davis, M., Rosenschein, S., Shapiro, N. [1982]. *Prospects and Problems for a General Modeling Methodology.* N-1801-RC. RAND, Santa Monica, CA.

de Kleer, J., Brown, J. [1984]. A qualitative physics based upon confluences. *Artificial Intelligence Journal*, Vol. 24, No. 7.

Evans, J. B. [1988]. *Structures of Discrete-Event Simulation. An Introduction to the Engagement Strategy.* Ellis Horwood, NY.

Fishman, G. [1973]. *Concepts and Methods in Discrete-Event Digital Simulation.* John Wiley & Sons, NY.

Forbus, K. [1984]. Qualitative process theory. *Artificial Intelligence Journal*, Vol. 24, pp. 85–168.

Fox, M. S., Sadeh, N., Baykan, C. [1989]. Constrained heuristic search. *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 309–316, Morgan Kaufmann, Los Altos, CA.

Galton, A. [1988]. *Temporal Logics and Their Applications.* Academic Press, NY.

Hanks, S., McDermott, D. [1987]. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, Vol. 33, No. 3, November, pp. 279–412.

Harel, D. [1984]. Statecharts: A visual approach to complex systems. *Proceedings of Advanced NATO Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* NATO ASI series F, Vol. 13, pp. 1–44, Springer Verlag, NY.

Hill, R. [1974]. *LUSH Resolution and Its Completeness.* DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh.

Hilton, M. L. [1987]. *ERIC: An Object-Oriented Simulation Language.* Rome Air Development Center, RADC-TR-87-103.

Ho, Y.-C. [1987]. Performance evaluation and perturbation analysis of discrete-event dynamic systems. *IEEE Transactions on Automatic Control,* Vol. AC-32, No. 7.

Hobbs, J., Moore, R. (eds.) [1985]. *Formal Theories of the Commonsense World.* Ablex Publishing Corporation, Norwood, NJ.

IEEE [1989]. Special issue on dynamics of discrete-event systems. *Proceedings of the IEEE,* January.

Inan K., Varaiya, P. [1987]. Finitely recursive processes. *Discrete Event Systems: Models and Applications.* Lecture Notes in Information Science, Vol. 103, Springer Verlag, NY.

Iwasaki, Y., Simon, H. [1986]. Causality in device behavior. *Artificial Intelligence,* Vol. 29.

Jahanian, F., Mok, A. [1986]. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering,* September.

Jefferson, D. [1985]. Virtual Time. *ACM Transactions on Programming Languages and Systems,* July.

Kiviat, P. J. [1967]. *Digital Computer Simulation: Modeling Concepts.* RM-5378-PR, RAND, Santa Monica, CA.

Klahr, P. [1985]. Expressibility in ROSS: An object-oriented simulation system. *AI APPLIED TO SIMULATION: Proceedings of the European Conference at the University of Ghent,* February 1985, Society for Computer Simulation, San Diego, 1986, pp. 136–139.

Klahr, P., McArthur, D., Narain, S. [1982]. SWIRL: An object-oriented air battle simulator. *Proceedings of the Second National Conference on Artificial Intelligence,* Pittsburgh, PA, pp. 331–334.

Klahr, P., Ellis, J., Giarla, W., Narain, S., Cesar, E., Turner, S. [1984]. *TWIRL: Tactical Warfare in the ROSS Language.* R-3158-AF, RAND, Santa Monica, CA.

Kowalski, R. [1979]. *Logic for Problem Solving.* Elsevier North Holland, NY.

Kowalski, R. [1986]. *Database Updates in the Event Calculus.* DoC 86/12, Department of Computing, Imperial College, London.

Kowalski, R., Sergot, M. [1986]. A logic-based calculus of events. *New Generation Computing*, Ohmsha Ltd., & Springer Verlag, NY, Vol. 4.

Kuipers, B. [1986]. Qualitative simulation. *Artificial Intelligence*, Vol. 29, pp. 289–338.

Kurshan, R. [1992]. Automata-Theoretic verification of coordinating processes. Technical Report, Mathematical Sciences Research Center, AT&T Bell Laboratories, Murray Hill, NJ, August.

Lamport, L. [1983]. What good is temporal logic? *Proceedings of IFIP*, R.E.A. Mason (ed.), Elsevier North Holland, NY.

Lloyd, J. W. [1984]. *Foundations of Logic Programming*, Springer Verlag, NY.

Manna, Z., Pnueli, A. [1981]. Temporal verification of concurrent programs. *The Correctness Problem in Computer Science*, R. S. Boyer, J. S. Moore (eds.), Academic Press, NY.

McArthur, D., Klahr, P., Narain, S. [1984]. *ROSS: An Object-Oriented Language for Constructing Simulations*. R-3160-AF, RAND, Santa Monica, CA.

McArthur, D., Klahr, P., Narain, S. [1985]. *The ROSS Language Manual*. N-1854-AF, RAND, Santa Monica, CA.

McCarthy, J. [1987]. Generality in artificial intelligence. Turing award lecture. *Communications of the ACM*, December.

McCarthy, J., Hayes, P. [1969]. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, B. Meltzer, D. Michie (eds.), Edinburgh University Press, Edinburgh.

McDermott, D. [1982]. A temporal logic for reasoning about processes and plans. *Cognitive Science*, Vol. 6, No. 2, April–June, pp. 101–155.

Misra, J. [1986]. Distributed discrete-event simulation. *Computing Surveys*, March.

Nance, R. [1981]. The time and state relationships in simulation modeling. *Communications of the ACM*, April.

Narain, S., Rothenberg, J. [1990]. Proving temporal properties of hybrid systems. *Proceedings of Winter Simulation Conference*, New Orleans, pp. 250–256, Society for Computer Simulation, San Diego, CA.

Narain, S., Rothenberg, J. [1989]. A logic for simulating dynamic systems. *Proceedings of Winter Simulation Conference*, Washington, D.C.

Ness, L. [1990]. Issues arising in the analysis of L.0. *Proceedings of the DIMACS: Computer-Aided Verification Workshop*, June 18–21.

Nugent, R. O. [1983]. *A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modeling*. WP-83W00407, The Mitre Corporation, McLean, VA.

Nugent, R. O., Wong, R. W. [1986]. The battlefield environment model: an army-level object-oriented simulation model. *The Proceedings of the 1986 Summer Simulation Conference*, Reno, NV.

O'Donnell, M. J. [1985]. Equational logic as a programming language. MIT Press, Cambridge, MA.

Ostroff, J. [1989]. Synthesis of controllers for real-time discrete-event systems. *IEEE Conference on Decision Control*, Tampa, FL.

Peterson, J. L. [1977]. Petri nets. *ACM Computing Surveys*, Vol. 9, No. 3.

Pnueli, A. [1977]. The temporal logic of programs. *Proceedings of 18th Annual IEEE Symposium on Foundations of Computer Science.*

Rothenberg, J. [1986]. Object-oriented simulation: Where do we go from here? *Proceedings of the 1986 Winter Simulation Conference* (Washington, D.C., December 8–10), J. Wilson, J. Henriksen, and S. Roberts (eds.), The Society for Computer Simulation, San Diego, CA, pp. 464–469.

Rothenberg, J., Narain, S., Steeb, R., Hefley, C., Shapiro, N. [1989]. *Knowledge-Based Simulation: An Interim Report.* N-2897-DARPA, RAND, Santa Monica, CA.

Sadeh, N., Fox, M. S. [1989]. *Preference Propagation in Temporal/Capacity Constraint Graphs.* Carnegie Mellon University Robotics Institute Report, CMU-RI-TR-89-2.

Sandewall, E. [1989]. Combining logic and differential equations for describing real-world systems. *Proceedings of International Conference on Knowledge Representation*, Toronto, Canada.

Schank, J., Leverich, B., Paul, J. [1990]. *Decision Support for the Wartime Theater Ammunition Distribution System: Research Accomplishments and Future Directions.* R-3794-A, RAND, Santa Monica, CA.

Schank, J., Mattock, M., Sumner, G., Greenberg, I., Rothenberg, J., Stucker, J. [1991]. *A Review of Strategic Mobility Models and Analysis.* R-3926-JS, RAND, Santa Monica, CA.

Schruben, L. [1983]. Simulation modeling with event graphs. *Communications of the ACM*, November.

Shoham, Y. [1987]. Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence Journal*, Vol. 33, pp. 89–104.

Suri, R. [1987]. Infinitesimal perturbation analysis for general discrete-event systems. *Journal of the ACM*, July.

Zeigler, B. [1984]. *Multifacetted modelling and discrete-event simulation.* Academic Press, NY.